DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

**PROYECTO FIN DE CARRERA**


# DETECCIÓN Y LOCALIZACIÓN DE OBSTÁCULOS EN ENTORNOS URBANOS MEDIANTE VISIÓN ESTÉREO

**AUTOR:** VIOLETA JIMÉNEZ MONJE
**TUTOR:** BASAM MUSLEH LANCIS

*A todos los que han hecho posible este trabajo.*

# ÍNDICE

ÍNDICE DE FIGURAS ..... i
ÍNDICE DE TABLAS ..... vi
LISTADO DE ABREVIATURAS ..... vii

1. RESUMEN ..... 1
ABSTRACT ..... 2
2. INTRODUCCIÓN ..... 3
2.1. MOTIVACIÓN DEL PROYECTO ..... 3
2.2. SISTEMAS DE TRANSPORTE INTELIGENTES ..... 5
2.3. PLATAFORMAS DE INVESTIGACIÓN ..... 8
2.3.1. IvvI ..... 8
2.3.2. iCab ..... 10
2.4. IMPLEMENTACIÓN DE LOS ADAS EN LA INDUSTRIA ..... 10
2.4.1. CONTINENTAL ..... 11
2.4.2. BMW ..... 12
2.4.3. FORD ..... 12
2.4.4. VOLVO ..... 13
2.4.5. VOLKSWAGEN ..... 14
2.4.6. GOOGLE ..... 15
3. FUNDAMENTOS TEÓRICOS ..... 17
3.1. PRINCIPIOS ÓPTICOS ..... 17
3.1.1. MODELO DE LENTE FINA ..... 17
3.1.2. MODELO DE PIN-HOLE ..... 18
3.1.3. SISTEMA ESTÉREO IDEAL ..... 19
3.1.4. GEOMETRÍA EPIPOLAR ..... 21
3.2. ELIMINACIÓN DEL RUIDO ..... 22
3.2.1. TIPOS DE RUIDO ..... 22
3.2.2. FILTROS FRECUENCIALES ..... 24
3.3. FILTROS ORIENTADOS A LA DETECCIÓN DE BORDES ..... 25

3.3.1. OPERADORES BASADOS EN LA PRIMERA DERIVADA (GRADIENTE) 25
3.3.2. OPERADORES BASADOS EN LA SEGUNDA DERIVADA (LAPLACIANA) 26
3.3.3. OPERADOR DE MARR-HILDRETH 27
3.4. MAPA DE DISPARIDAD 27
3.4.1. CÁLCULO DE LA FUNCIÓN DE COSTE 30
3.4.2. AGREGACIÓN DEL COSTE 30
3.4.3. CÁLCULO DE LA DISPARIDAD 31
3.4.4. POST-PROCESAMIENTO 32
3.5. U-V DISPARITY 33
3.6. TRANSFORMADA DE HOUGH 34
3.7. BLOB ANALYSIS 35
4. DESARROLLO DEL ALGORITMO 37
4.1. MAPA DE DISPARIDAD 37
4.1.1. VERSIÓN 1 38
4.1.2. VERSIÓN 2 39
4.1.3. VERSIÓN 3 41
4.1.4. VERSIÓN 4 43
4.1.5. VERSIÓN 5 48
4.2. DETECCIÓN DE OBSTÁCULOS 51
4.2.1. U_DISPARITY 51
4.2.2. MAPA DE OBSTÁCULOS 52
4.2.3. DETERMINACIÓN DE LAS REGIONES DE INTERÉS SOBRE LOS OBSTÁCULOS 52
4.3. LOCALIZACIÓN DE OBSTÁCULOS 54
4.3.1. MAPA LIBRE 55
4.3.2. OBTENCIÓN DEL *ROAD PROFILE* A PARTIR DE LA *V_DISPARITY* 56
4.3.3. LOCALIZACIÓN DE OBSTÁCULOS MEDIANTE LA DISPARIDAD 59
4.3.4. LOCALIZACIÓN DE OBSTÁCULOS MEDIANTE EL *ROAD PROFILE* 59
4.3.5. RESULTADOS EXPERIMENTALES 62
5. RESULTADOS 65
5.1. ANÁLISIS DEL ALGORITMO DE CÁLCULO DE LA DISPARIDAD 65
5.1.1. VERSIONES 4 Y 5 66

5.2. ANÁLISIS DEL ALGORITMO DE DETECCIÓN Y LOCALIZACIÓN DE OBSTÁCULOS ..... 70
6. CONCLUSIONES Y TRABAJOS FUTUROS ..... 73
6.1. CONCLUSIONES ..... 73
6.2. TRABAJOS FUTUROS ..... 73
7. COSTES DEL PROYECTO ..... 74
8. BIBLIOGRAFÍA ..... 75
ANEXO 1 · CÓDIGO DEL ALGORITMO ..... 78
1. TRATAMIENTO DE IMÁGENES CON LAS LIBRERÍAS ..... 78
1.1. LIBRERÍAS MIL ..... 78
1.2. LIBRERÍAS OpenCV ..... 79
2. CÓDIGO DEL ALGORITMO ..... 80
ANEXO 2 · EVOLUCIÓN DEL ALGORITMO DEL CÁLCULO DE LA DISPARIDAD ..... 94

# ÍNDICE DE FIGURAS

Figura 2.1 Accidentes con víctimas en España desde 1985 [3] ..... 3

Figura 2.2 Accidentes con víctimas en España desde 1985 por cada 1.000 vehículos [3] ..... 3

Figura 2.3 Peatones víctimas de accidentes de tráfico en España desde 1985 [3] ..... 4

Figura 2.4 Peatones víctimas de accidentes de tráfico en España desde 1985 por cada 1.000 vehículos [3] ..... 4

Figura 2.5 Vehículo diseñado por la universidad Carnegie Mellon participante en el DARPA Grand Challenge 2004 [1] ..... 7

Figura 2.6 Vehículo diseñado por la universidad Carnegie Mellon ganador del Urban Challenge 2007 [15] ..... 8

Figura 2.7 Plataforma de investigación IvvI ..... 9

Figura 2.8 Plataforma de investigación iCab ..... 10

Figura 2.9 Fuentes de recogida de datos y sus radios de acción [17] ..... 15

Figura 3.1 Modelo de lente fina ..... 18

Figura 3.2 Modelo de cámara *pin-hole* ..... 18

Figura 3.3 Parámetros del modelo *pin-hole*, (a) vista aérea, (b) vista lateral ..... 19

Figura 3.4 Modelo *pin-hole* para un sistema estéreo ..... 20

Figura 3.5 Geometría epipolar ..... 21

Figura 3.6 Geometría epipolar para una imagen normalizada ..... 21

Figura 3.7 (a) Imagen original, (b) imagen afectada por ruido gaussiano ..... 22

Figura 3.8 (a) Imagen original, (b) imagen afectada por ruido impulsional ..... 23

Figura 3.9 (a) Imagen original, (b) Imagen afectada por ruido frecuencial ..... 23

Figura 3.10 (a) Imagen original, (b) imagen afectada por ruido multiplicativo ..... 24

Figura 3.11 (a), (b) Imágenes izquierda y derecha [20]; (c), (d) mapas de disparidad; (e), (f) mapas de disparidad equalizados ..... 29

Figura 3.12 (a) Disparidad de la imagen izquierda con respecto a la derecha (imagen equalizada), (b) disparidad de la imagen derecha con respecto a la izquierda (imagen equalizada), (c) disparidad resultante tras el *cross-checking*, (d) disparidad resultante tras el *cross-checking* (imagen equalizada) ..... 33

Figura 3.13 Mapa de disparidad (imagen equalizada), (derecha) *v_disparity*, (abajo) *u_disparity* ..... 34

Figura 3.14 Representación gráfica de los parámetros paramétricos y cartesianos ..... 35

Figura 4.1 Evolución del algoritmo de cálculo de la disparidad ..... 37

Figura 4.2 Proceso de comparación entre las dos imágenes estéreo ..... 38

Figura 4.3 Esquema del cálculo del coste para una ventana centrada en el píxel ($x_0$,$y_0$+1) utilizando el coste de la ventana centrada en ($x_0$,$y_0$) ..... 40

Figura 4.4 (a) Detalle de la imagen izquierda, donde se muestran diversas ventanas (nombradas como A,B,C...); (b) detalle de la imagen derecha, donde se muestran diversas ventanas (a,b,c...) ..... 42

Figura 4.5 (a) Valores constitutivos del vector coste para una posición del mapa de disparidad de la imagen izquierda con respecto a la derecha, en función de la nomenclatura indicada en la figura 4.4, para una $d_{máx}$=3; (b) valores constitutivos de los vectores coste para diversas posiciones de la disparidad de la imagen derecha con respecto a la izquierda, con los valores coincidentes con el coste de la disparidad de la imagen izquierda con respecto a la derecha sombreados ..... 42

Figura 4.6 Solapamiento entre dos ventanas con la coordenada *x* correlativa ..... 44

Figura 4.7 Dimensiones de la matriz coste_fila, donde tam_y es el tamaño vertical de las imágenes de partida y $d_{máx}$ el valor máximo de la disparidad ..... 44

Figura 4.8 Solapamiento lateral entre dos ventanas con la coordenada *x* correlativa ..... 45

Figura 4.9 Esquema del cálculo del coste de agregación para la ventana centrada en $(x_0,y_0+1)$ ..... 46

Figura 4.10 Dimensiones de la matriz AD, donde tam_y es el tamaño vertical de las imágenes de partida y tam_x el horizontal ..... 47

Figura 4.11 Solapamiento entre dos ventanas con la coordenada x correlativa ..... 48

Figura 4.12 Solapamiento lateral entre dos ventanas con la coordenada y correlativa ..... 48

Figura 4.13 Solapamiento entre ventanas ..... 49

Figura 4.14 Detalle del solapamiento entre ventanas ..... 49

Figura 4.15 Dimensiones y distribución de la matriz de coste ..... 50

Figura 4.16 (a) Mapa de disparidad (imagen equalizada), (b) *u_disparity*, (c) *u_disparity* umbralizada ..... 51

Figura 4.17 Mapa de obstáculos (imagen equalizada) ..... 52

Figura 4.18 (a) Resultado del *Blob Analysis*: identificación de tres obstáculos en la imagen, (b) regiones de interés determinadas ..... 53

Figura 4.19 (a) Mapa de obstáculos (imagen equalizada), (b) bordes presentes en el mapa de obstáculos, (c) resultado del *Blob Analysis*: identificación de cuatro obstáculos en la imagen, (d) regiones de interés determinadas ..... 54

Figura 4.20 (a) Mapa de disparidad (imagen equalizada), (b) mapa libre (imagen equalizada) ..... 55

Figura 4.21 (b) *v_disparity* calculada a partir del mapa de disparidad (a); (d) *v_disparity* calculada a partir del mapa libre (c) ..... 56

Figura 4.22 (a) *v_disparity*, (b) *v_disparity* con la recta resultante de la transformada de Hough ..... 57

Figura 4.23 Ejemplos de obtención del *road profile* en diferentes casos en entornos urbanos: (a) vehículo y pared, (b) atasco, (c) obstáculo a ambos lados, (d) ejemplo de obstáculo elevado: un túnel. La línea roja marca el road profile identificado en cada caso ..... 58

Figura 4.24 (a) Mapa de disparidad, (b) *road profile* ..... 60

Figura 4.25 Esquema de la estimación de la $d_{teórica}$ ..... 60

Figura 4.26 Interpolación entre el mapa de disparidad y el *road profile* ..... 60

Figura 4.27 Relación entre las distancias cámara-objeto y coche-objeto ..... 61

Figura 4.28 Parámetros del modelo *pin-hole* para un punto P(U,V,W) perteneciente al suelo ..... 62

Figura 4.29 Comparación de la resolución de la posición usando los valores de la disparidad (línea superior) y usando el *road profile* (inferior) ..... 62

Figura 4.30 Test de localización de obstáculos en un entorno urbano. (a) Diversas imágenes de un peatón realizando un recorrido en zig-zag frente al vehículo; (b) resultados de su localización (en rojo, mediante el método de la disparidad; en azul, a través del *road profile*); (c) recorrido de dos peatones que se cruzan frente al vehículo; (d) resultados de la localización para el cruce de dos vehículos ..... 63

Figura 5.1 Tiempo de cómputo de las diversas versiones del algoritmo de cálculo del mapa denso de disparidad ..... 66

Figura 5.2 Tiempo de cómputo de cada una de las versiones en función del tamaño de ventana para diversos tamaños de imagen ..... 67

Figura 5.3 Mapa denso de disparidad calculado para un tamaño de ventana de (a) 11x11px, (b)17x17px. y (c)23x23px (imágenes equalizadas) ..... 68

Figura 5.4 Tiempo de cómputo de las versiones 4 y 5 variando la disparidad máxima para diversos tamaños de imagen ..... 69

Figura 5.5 Tiempo de cómputo de cada una de las versiones en función del tamaño de la imagen ..... 69

Figura 5.6 Tiempo de cómputo de cada una de las versiones en función del tamaño de la imagen (en px) ..... 70

Figura 5.7 Tiempo de cómputo en función del tamaño de ventana para diversos tamaños de imagen ..... 71

Figura 5.8 Tiempo de cómputo en función del tamaño de la imagen ..... 71

Figura 5.11 Tiempo de cómputo en función del tamaño de la imagen (en px) ..... 72

# ÍNDICE DE TABLAS

Tabla 5.1 Tiempo de cómputo de cada una de las versiones del algoritmo de cálculo del mapa denso de disparidad (en segundos) donde, en las versiones 4 y 5 optimizadas, se calculan todas las AD al principio del algoritmo ..... 65

Tabla 5.2 Tiempo de cómputo de la versión 4 optimizada del algoritmo de cálculo del mapa denso de disparidad (en segundos) en función de diversos parámetros (tamaño de imagen, tamaño de ventana, $d_{máx}$) ..... 66

Tabla 5.3 Tiempo de cómputo de la versión 5 optimizada del algoritmo de cálculo del mapa denso de disparidad (en segundos) en función de diversos parámetros (tamaño de imagen, tamaño de ventana, $d_{máx}$) ..... 67

Tabla 5.4 Tiempo de cómputo del algoritmo de detección y localización de obstáculos (en segundos) en función de diversos parámetros para una disparidad máxima de 30 píxeles ..... 70

# LISTADO DE ABREVIATURAS

| | |
|---|---|
| AD | **A**bsolute Intensity **D**ifferences |
| ADAS | **A**dvanced **D**river **A**ssistance **S**ystem (Sistemas Avanzados de Ayuda a la Conducción) |
| AHSRA | **A**dvanced Cruisse-Assist **H**ighway **S**ystem **R**esearch **A**ssociation |
| Blob | **B**inary **L**arge **Ob**ject |
| CWAB | **C**ollision **W**arning with **A**uto **B**rake |
| DAC | **D**river **A**lert **C**ontrol |
| DARPA | **D**efense **A**dvanced **R**esearch **P**rojects **A**gency |
| ESC | **E**lectronic **S**tability **C**ontrol |
| HAVEit | **H**ighly **A**utomated **Ve**hicles for **I**ntelligent **T**ransport |
| iCab | **I**ntelligent **C**ampus **A**utomo**b**ile |
| ITS | **I**ntelligent **T**ransport **S**ystems (Sistemas de Transporte Inteligentes) |
| IvvI | **I**ntelligent **V**ehicle based on **V**isual **I**nformation |
| LDW | **L**ane **D**eparture **W**arning |
| lidar | **Li**ght **D**etection **a**nd **R**anging |
| NAHSC | **N**ational **A**utomated **H**ighway **S**ystem **C**onsortium |
| PROMETHEUS | **PRO**gra**M** for a **E**uropean **T**raffic with **H**ighest **E**fficiency and **U**nprecedented **S**afety |
| ROI | **R**egion **o**f **I**nterest |
| SD | **S**quared Intensity **D**ifferences |
| SLAM | **S**imultaneous **L**ocalization **a**nd **M**apping |
| SSD | **S**um of **S**quared **D**ifferences |
| TAP | **T**emporary **A**uto **P**ilot (Piloto Automático Temporal) |
| WTA | **W**inner **T**ake **A**ll |

| | |
|---|---|
| GPU | **G**raphics **P**rocessing **U**nit |

# 1. RESUMEN

Actualmente la industria automovilística centra sus esfuerzos en el desarrollo de sistemas de seguridad. Mientras algunos proyectos persiguen una total automatización de la conducción, la mayoría se centran en el desarrollo de sistemas de ayuda a la conducción.

En el presente trabajo se detalla un algoritmo capaz de detectar y localizar obstáculos a partir de las imágenes captadas por un par de cámaras estéreo que emulan el sistema de visión humano. En primer lugar se obtiene la información de la profundidad mediante la comparación de ambas imágenes. Posteriormente se usan estos datos para detectar los obstáculos que se situados delante del vehículo y determinar su localización de manera precisa.

Una vez desarrollado el algoritmo, se estudian los parámetros que lo configuran, así como su eficiencia en la detección y localización de obstáculos. El objetivo es evaluar la relación entre el coste computacional (tiempo) y la eficiencia de la solución obtenida.

**Palabras clave:**

Detección de obstáculos, Visión estéreo, *U-V disparity*.

## ABSTRACT

Nowadays the automobile industry is focusing its efforts on the development of security systems. While some projects pursue full automation of the driving, most focus on the develop of driver assistance systems.

This paper details an algorithm able to detect and locate obstacles from the images captured by a stereo rig that emulates the human vision system. Firstly the information about the depth is obtained through the comparison between both images. Subsequently this information is used to detect the obstacles in front of the vehicle and determine their location with a high resolution.

Once developed the algorithm, its forming parameters are studied, as well as its efficiency in the detection and location of obstacles. The objective is to evaluate the relationship between the computational cost (time) and the efficiency of the solution obtained.



**Keywords:**

Obstacle detection, Stereo vision, U-V disparity.

# 2. INTRODUCCIÓN

## 2.1. MOTIVACIÓN DEL PROYECTO

Actualmente, los esfuerzos de las compañías automovilísticas se centran en la mejora de la seguridad de la conducción.

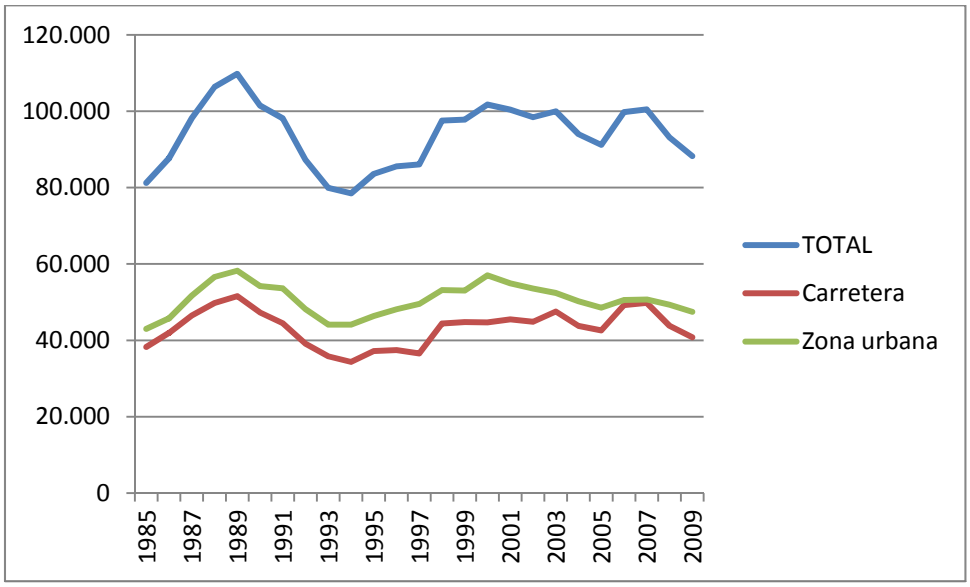



Figura 2.1 Accidentes con víctimas en España desde 1985 [3].

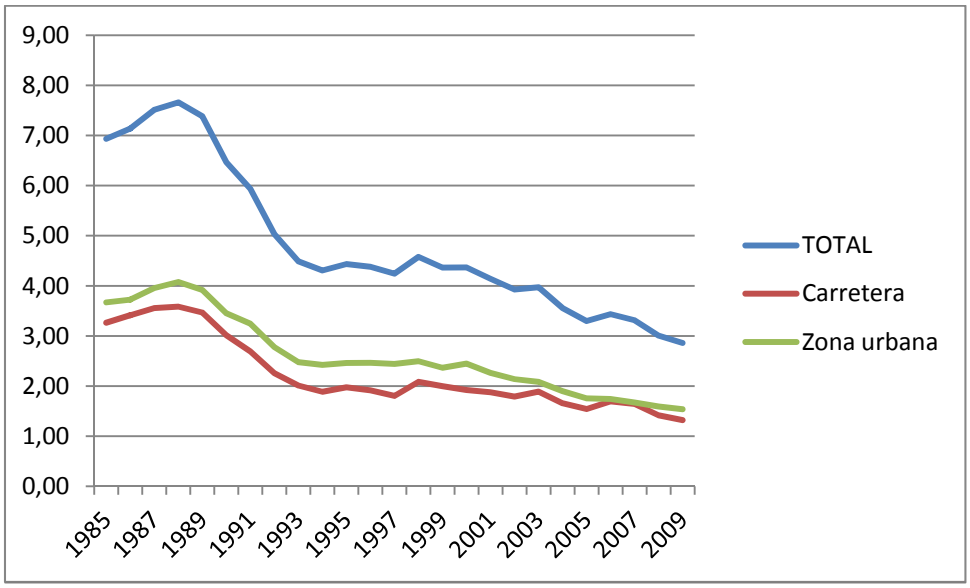



Figura 2.2 Accidentes con víctimas en España desde 1985 por cada 1.000 vehículos [3].

En el primer gráfico se aprecia que el número de accidentes con víctimas (heridos y/o muertos) en el periodo estudiado se ha mantenido constante. Sin embargo, teniendo en cuenta el aumento del parque de vehículos (que ha pasado de 11 millones en 1985 a más de 30 en 2009), se nota una reducción del 40% en la siniestralidad. Esta disminución se ha conseguido gracias a medidas a nivel estatal (mejora de las infraestructuras, campañas de concienciación, etc.).

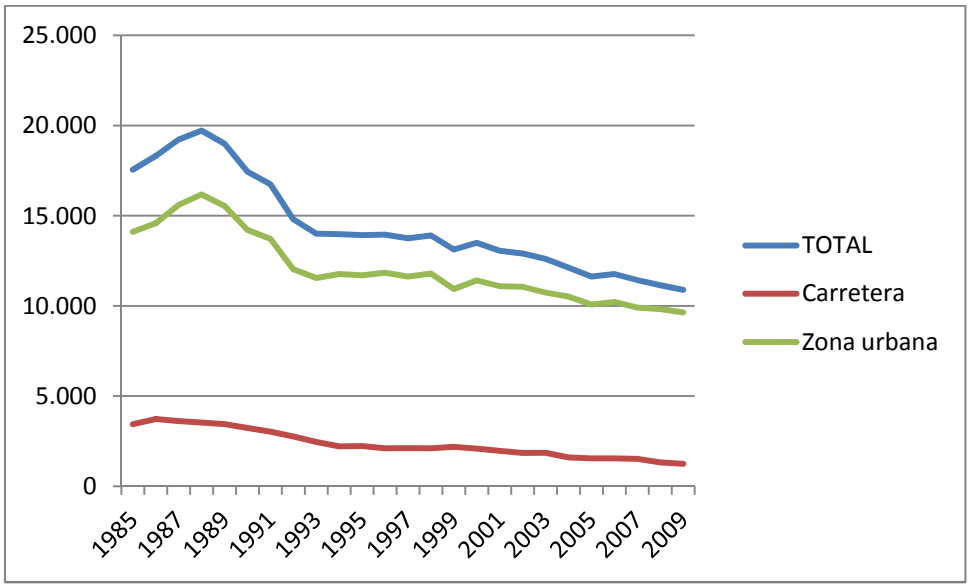



Figura 2.3 Peatones víctimas de accidentes de tráfico en España desde 1985 [3].

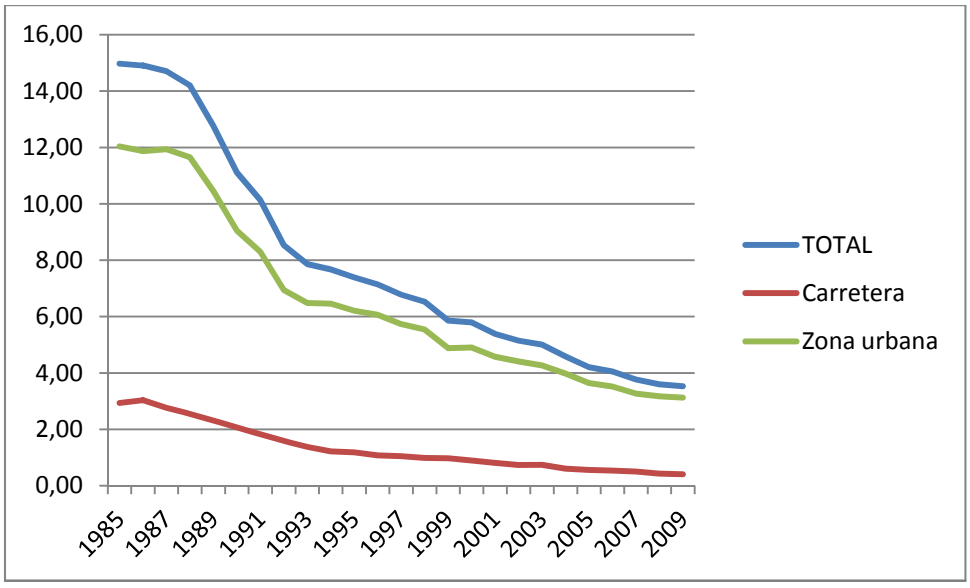



Figura 2.4 Peatones víctimas de accidentes de tráfico en España desde 1985 por cada 1.000 vehículos [3].

El descenso de peatones víctimas de accidentes sigue en gran medida la tendencia marcada por la siniestralidad. Si bien en datos absolutos se observa una disminución

de los viandantes víctimas (heridos y/o muertos) de accidentes, es en los datos relativos al número de vehículos donde se aprecia un clarísimo descenso.

Notar que si bien en la última década los elementos de seguridad pasiva en los vehículos (cinturones de seguridad, reposacabezas, airbags, etc.) se han universalizado, éstos están diseñados principalmente para proteger a los ocupantes del vehículo. De tal forma que el número de peatones víctimas de atropellos urbanos sigue siendo elevado (más de 9.500 casos al año [3]). Enfocando a resolver esta casuística, el sistema de detección de obstáculos diseñado está especialmente orientado a entornos urbanos.

## 2.2. SISTEMAS DE TRANSPORTE INTELIGENTES

La universalización del transporte por carretera acontecida en la segunda mitad del siglo XX trajo consigo una serie de retos: la mejora de la seguridad, la optimización de la red vial y la reducción del consumo energético y de los niveles de polución. En respuesta a estas necesidades surgió la idea de la conducción automática (completa o parcial). Las metas a alcanzar eran, entre otras, la detección de las líneas de carril, el mantenimiento de la distancia de seguridad, la regulación de la velocidad en función de las condiciones del tráfico y de las características de la vía, la asistencia al cambio de carril al adelantar o sortear un obstáculo, la optimización de la ruta y la ayuda a la circulación y el aparcamiento en entornos urbanos.

De este modo, hace 30 años, nacieron los Sistemas de Transporte Inteligentes *(ITS ≡ Intelligent Transport System)* con el objetivo de solucionar los problemas de movilidad de personas y mercancías.

En Europa, en 1986, se pone en marcha el proyecto PROMETHEUS (*PROgraM for a European Traffic with Highest Efficiency and Unprecedented Safety*) [4]. Integrado por 13 empresas fabricantes de vehículos y diversas unidades de investigación procedentes de gobiernos y universidades de 19 países.

En un primer momento, en EEUU, los estudios sobre movilidad se llevaron a cabo por diversas universidades, centros de investigación y compañías automovilísticas de forma desorganizada. Hasta 1995 no se creó la NAHSC (*National Automated Highway System Consortium*) [4].

Por su parte, en Japón, donde el problema es y sigue siendo acuciante, se llegaron a crear diversos prototipos antes de la fundación, en 1996, de la AHSRA (*Advanced Cruisse-Assist Highway System Research Association*) [4].

Entre los resultados de estas primeras investigaciones se puede citar el análisis en profundidad del problema, así como la creación de planes de viabilidad (requisitos de los sistemas y posibles efectos en la red vial).

En 2000 se comienza una nueva fase en los ITS. Gracias a la madurez del enfoque y a las nuevas posibilidades tecnológicas, comienzan a crearse los primeros productos experimentales. Dado su elevado coste, se trataba generalmente aplicaciones militares. El incremento del interés en estos sistemas, así como la mejora de la producción integral, contribuyó a su expansión a otros ámbitos.

En 2004, la DARPA (*Defense Advanced Research Projects Agency*), una Agencia dependiente del Departamento de Defensa de EEUU, organizó el primer DARPA Grand Challenge. Una competición para vehículos autónomos donde, el primero en completar el recorrido de 230 km. a través del desierto de Mojave, ganaría un millón de dólares [1].

Tras una década de inversiones fallidas en el desarrollo de vehículos autónomos, el ejército buscaba así atraer a investigadores, tanto del ámbito académico como autodidactas. Los participantes no tenían obligación de compartir los desarrollos presentados, sin embargo, el director de la DARPA, el Coronel Jose Negron, auguraba subvenciones para aquellas ideas más innovadoras. La motivación final del proyecto era alcanzar el objetivo, marcado por el Departamento de Defensa, de automatizar un tercio de los transportes militares para 2015.


Figura 2.5 Vehículo diseñado por la universidad Carnegie Mellon participante en el DARPA Grand Challenge 2004 [1].

En esta primera edición, ninguno de los 15 participantes terminó la prueba. El vehículo diseñado por la universidad Carnegie Mellon, un Hummer cuyo coste de construcción se estima en 3 millones de dólares, fue el último en abandonar, tras haber recorrido únicamente 11,3 km.

En 2005 se volvió a celebrar el DARPA Grand Challenge [32]. En esta ocasión tomaron la salida 23 vehículos, de los cuales 5 llegaron a meta. El ganador fue el prototipo de la universidad de Stanford, un Volkswagen Touareg equipado con cinco sensores láser en el techo para la detección de obstáculos en un rango de hasta 30 m. y una cámara de vídeo a color para distancias de hasta 50 m. La combinación de ambos dispositivos posibilitó que el vehículo completará la prueba con una velocidad media de 30,7 km/h (alcanzando velocidades puntas de 65 km/h).


Figura 2.6 Vehículo diseñado por la universidad Carnegie Mellon ganador del Urban Challenge 2007 [15].

En 2007, la última edición de la prueba hasta la fecha, se celebró el Urban Challenge [15]. En esta ocasión, los 11 vehículos participantes tenían que completar un recorrido de 96,5 km., obedeciendo las reglas de tráfico e interactuando con otros vehículos. El ganador, el prototipo diseñado por la universidad Carnegie Mellon, fue elegido por su combinación de velocidad y estilo de conducción.

## 2.3. PLATAFORMAS DE INVESTIGACIÓN

El Laboratorio de Sistemas Inteligentes de la Universidad Carlos III dispone de dos modelos distintos de vehículos donde testar las implementaciones desarrolladas en dicho área.

### 2.3.1. Ivvl

El Ivvl (*Intelligent Vehicle based on Visual Information*) es una plataforma de investigación para el desarrollo de Sistemas Avanzados de Ayuda a la Conducción (*ADAS* ≡ *Advanced Driver Assistance System*). Se trata de un Nissan Note (Figura a) donde actualmente se han implementado diversos sistemas basados en la visión por ordenador y tecnología láser. El objetivo es probar estos sistemas en entornos urbanos reales.

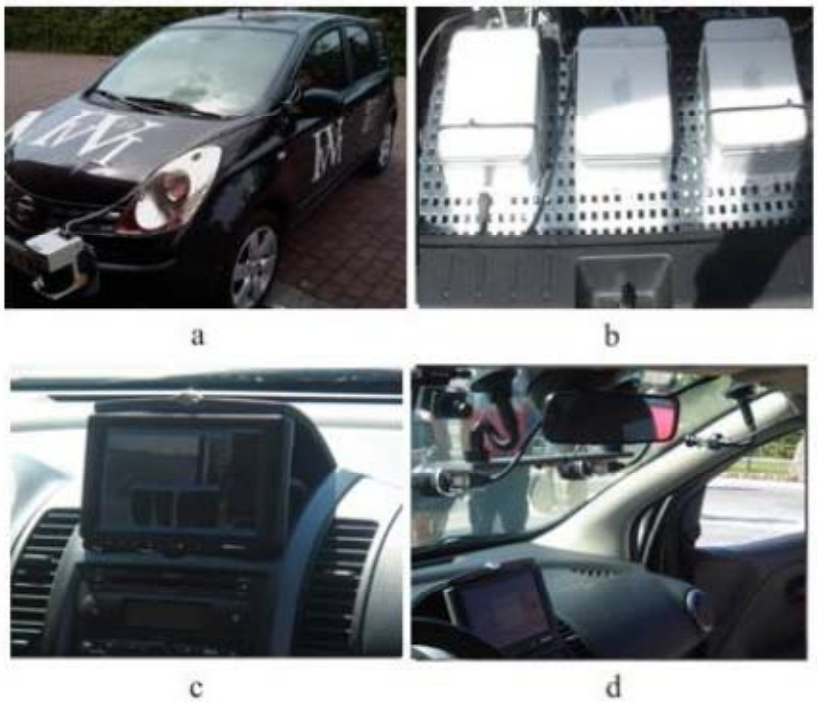



Figura 2.7 Plataforma de investigación Ivvl.

Para ello se dispone de un convertidor de potencia DC/AC conectado a una batería auxiliar que permite alimentar los siguientes dispositivos:

- Un monitor (Figura c), similar a un GPS, donde se muestra, en tiempo real, la posición de los obstáculos encontrados. Además, a través de los altavoces del vehículo, se emiten diversos tipos alertas.
- Tres PCs (Figura b), ubicados en el maletero del vehículo, usados para procesar la información captada por los sensores.
- Dos cámaras a color que detectan, respectivamente, la señalización vertical de la vía y el estado de somnolencia del conductor.
- Un sistema de visión binocular para la detección de obstáculos durante la conducción diurna (Figura d). La interpretación de estas imágenes es el objetivo del presente proyecto.
- Una cámara infrarroja para la detección de peatones en condiciones de conducción nocturna.
- Un sensor láser, situado en el parachoques delantero, para la detección y clasificación de peatones y vehículos.

### 2.3.2. iCab

El iCab (*Intelligent Campus Automobile*) (figura 2.8) es un vehículo eléctrico diseñado para el transporte de personas de forma autónoma dentro del campus.

Figura 2.8 Plataforma de investigación iCab.

Para poder ser controlado por un ordenador de abordo, ha sido modificado mecánica y electrónicamente.

## 2.4. IMPLEMENTACIÓN DE LOS ADAS EN LA INDUSTRIA

Actualmente la industria automovilística está centrada en el desarrollo de ADAS como medida de reducción de la siniestralidad. A pesar de que la implantación de este tipo de sistemas aún no sea generalizada, la mayoría de las marcas comercializan vehículos que incluyen diversas versiones de este tipo de equipamiento. Además de los fabricantes, otras compañías, relacionadas (como Continental) o no (por ejemplo, Google), con el ámbito automovilístico están llevando a cabo sus propios desarrollos.

A continuación se enuncian, ordenados en función del grado de automatización de la conducción, algunos de los proyectos desarrollados por diversas compañías en el ámbito de los ADAS. No se busca realizar una compilación exhaustiva de todos los sistemas existentes sino ofrecer una visión general del estado del arte.

### 2.4.1. CONTINENTAL

El fabricante de neumáticos (y otros componentes para el automóvil) alemán Continental ha desarrollado un sistema ADAS denominado ContiGuard [8]. El objetivo es mejorar la seguridad en la conducción mediante la integración de sistemas activos y pasivos, reduciendo el número de accidentes y minimizando sus consecuencias para todos los usuarios de la vía.

ContiGuard distingue cinco etapas en la conducción, en cada una de las cuales se activan los sistemas correspondientes:

- Conducción normal: Permanecen activas diversas funciones de apoyo a la conducción: control de crucero adaptativo, control de dirección y control de límite de velocidad.

- Conducción en situaciones de peligro: En estas circunstancias se activan diversas funciones de advertencia como la de distancia de seguridad, la de cambio de carril o la de exceso de velocidad. En caso necesario, se activa el ESC (*Electronic Stability Control*) para estabilizar el vehículo. También entran en funcionamiento mecanismos de asistencia a la frenada, desde el precarga de los frenos al frenado autónomo de emergencia.

- Conducción pre-accidente: Se activan funciones de protección adicionales como la tensión del cinturón de seguridad.

- Conducción durante el accidente: El objetivo es ofrecer la máxima protección en caso de accidente inminente, buscando mitigar el impacto y sus consecuencias. Una de las principales medidas de protección es el airbag.

- Conducción post-accidente: Se realiza de forma automática la llamada a urgencias solicitando asistencia.

### 2.4.2. BMW

BMV actualmente comercializa un sistema de seguridad llamado **ConnectedDrive** [6]**.** Inicialmente se trataba de una aplicación telemática capaz de localizar el vehículo y comunicar al conductor información en tiempo real sobre el estado de la vía (atascos, condiciones meteorológicas, accidentes, etc.). Además, en caso de accidente, realiza de forma automática una llamada a urgencias informando de la ubicación del siniestro.

Este sistema se ha ampliado con nuevas funcionalidades, en el campo de visión artificial destacan tres:

- Parking Assistant: Para velocidades inferiores a 35 Km/h este asistente encuentra sitio para estacionar (de una longitud 1,2 m mayor que el vehículo). Además, a la hora de realizar la maniobra asume el control del volante, quedando el acelerador y el freno a cargo del conductor.

- Surround View: A partir de cinco microcámaras (una trasera y dos en cada retrovisor) se genera una imagen de las zonas lateral y trasera del vehículo que se muestra en el monitor de abordo.

- BMW Night Vision: Mediante una cámara infrarroja se muestran personas y animales en el display frontal.

### 2.4.3. FORD

Por su parte, Ford centra sus esfuerzos en el campo de la prevención de siniestros en el desarrollo de dos sistemas complementarios:

- Ford Active City Stop [11]: Es un sistema diseñado para evitar o mitigar accidentes a velocidades inferiores a 30 km/h. Un sensor lidar (*Light Detection and Ranging*) [25] situado en la zona alta del luneta delantera barre un área de hasta 7,6 m delante del vehículo en busca de obstáculos. Si se detecta un automóvil parado, moviéndose lentamente o frenando, se evalúa la situación como una colisión inminente, los frenos se

precargan para responder con mayor rapidez. Si el conductor no actúa (reduciendo la aceleración o frenando) el coche frena automáticamente y desactiva el acelerador.

- Ford Lane Keeping Aid [12]: El sistema de permanencia en el carril utiliza la cámara montada en la parte superior del parabrisas, analizando en tiempo real las imágenes tomadas para detectar las marcas de delimitación de los carriles. Cuando el vehículo comienza a acercarse a las marcas viales se lleva a cabo una ligera corrección automática de la dirección con la finalidad de informar al conductor. Si se detecta la salida del carril, el aviso se transmitirá en forma de vibración del volante.

### 2.4.4. VOLVO

Actualmente Volvo oferta diversos tipos de equipamientos para la seguridad preventiva [34]. Al encontrarse en las primeras etapas de implantación, este tipo de sistemas sólo están disponibles en modelos de alta gama.

- City Safety: Tecnología diseñada para evitar colisiones a velocidades de hasta 30 km/h. Un dispositivo láser detecta si el vehículo de delante está parado o se mueve lentamente (a menos de 15 km/h). Si se identifica una posible colisión inminente, los frenos se precargan para responder con mayor rapidez. Si el conductor no frena, el City Safety aplica automáticamente los frenos y desconecta el acelerador para mitigar los efectos de una colisión.

- CWAB (*Collision Warning with Auto Brake*): Tecnología para la advertencia de colisión con freno automático y detección de peatones. Con un alcance de 150 metros, un sensor radar situado tras la parrilla y una cámara digital detrás del parabrisas monitorizan continuamente la distancia al vehículo que circula delante. Durante la luz del día, a velocidades de hasta 35 km/h, también detecta peatones parados o en movimiento. El sistema avisa al conductor si está cerca del vehículo situado delante o si hay peatones en su camino. Si no se reacciona y la colisión es inminente, se aplica el freno de forma automática para que evitar o mitigar el impacto.

- DAC (*Driver Alert Control*): Se trata de un sistema de alerta del conductor. Se compara la información extraída de una cámara que monitoriza el tramo de carretera delantero y los datos de los movimientos del volante con un patrón de conducción normal, detectando cuándo éste se vuelve errático. En ese momento se emite una señal acústica y la recomendación de parar a descansar aparece en el panel de instrumentos.

- LDW (*Lane Departure Warning*): Advertencia de cambio de carril. Para velocidades a partir de 65 Km/h entra en funcionamiento este sistema que mediante una cámara registra las marcas de la calzada y comprueba la posición del vehículo en la misma. En caso de una salida del carril no voluntaria (sin activar los intermitentes), el sistema emite una señal acústica.

### 2.4.5. VOLKSWAGEN

El proyecto **HAVEit** (*Highly Automated Vehicles for Intelligent Transport*) [19], creado en 2008, ha sido desarrollado por un consorcio europeo formado por compañías automovilísticas, compañías de repuestos e institutos científicos, con un presupuesto total de 28 millones de euros.

En junio de 2011, Volkswagen presentó el resultado del proyecto: el Piloto Automático Temporal (*TAP* ≡ *Temporary Auto Pilot*). El vehículo circula semi-automáticamente, con supervisión del conductor, a velocidades de hasta 130 Km/h. El objetivo es prevenir errores debidos a la pérdida de concentración del conductor [18]. Para ello, el sistema mantiene la distancia de seguridad, conduciendo a la velocidad seleccionada por el conductor y reduciéndola de acuerdo a las condiciones de la vía, mantiene al vehículo centrado en el carril, respetando las normas de adelantamiento, los límites de velocidad, realiza las maniobras de detención y arranque durante los atascos.

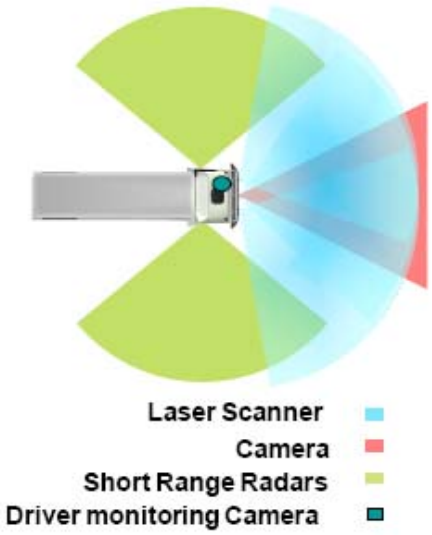



Figura 2.9 Fuentes de recogida de datos y sus radios de acción [17].

Como se puede ver en la imagen previa, los vehículos HAVEit cuentan con cuatro tipos de sensores para la adquisición de los datos [17]. La zona frontal es cubierta por cámaras y, en un ángulo más amplio, por escáneres láser; los laterales por sensores radar de corto alcance y, además, se dispone de una cámara interior encargada de monitorizar el estado del conductor.

### 2.4.6. GOOGLE

Google ha diseñado un vehículo totalmente autónomo, el cual ha recorrido 250.000 kilómetros por calles y carreteras de Estados Unidos a velocidades de hasta 100 Km/h [25]. Para garantizar la seguridad, viajaron en el vehículo un conductor y un ingeniero, no siendo necesaria su actuación en ningún momento.

Para ello, ha equipado una flota de vehículos (6 Toyota Prius y un Audi TT) [9] con radares, cámaras de vídeo, sensores de movimiento y dispositivos GPS. Toda la información recogida, además de servir como base a la conducción autónoma, se muestra a los pasajeros a través de una pantalla a color en tres dimensiones de 22 pulgadas situada en el salpicadero. En ella se muestra el entorno con gran detalle, representando las marcas del suelo, las señales verticales y los semáforos. El carril en el que se encuentra el vehículo aparece señalado con una franja verde. Y, en caso de adelanta-

miento, el coche a sobrepasar se representa con un bloque amarillo, mientras una voz informa de la maniobra.

El principal sensor del sistema va montado en la parte central del techo. Conocido como lidar, proporciona un mapa tridimensional del entorno de radio hasta 70 m con una precisión de centímetros. Además, cada vehículo se ha equipado con cuatro sensores radar estándar (tres en la zona delantera y uno en la parte posterior), con menor resolución pero mayor alcance que los lidar. Dentro del vehículo, junto al espejo retrovisor, se ha instalado una cámara de video de alta definición para la detección de semáforos y obstáculos en movimiento, como peatones y ciclistas. Por último, se ha equipado a los vehículos con un sistema GPS y un sensor inercial de movimiento.

La técnica de navegación usada por los vehículos autónomos diseñados por Google se conoce como SLAM (*Simultaneous Localization and Mapping*). Partiendo de un mapa del entorno previo, el vehículo lo construye y actualiza, ubicándose en él. Para confeccionar dicho mapa SLAM hay que recorrer una primera vez la ruta en conducción manual, permitiendo a los sensores capturar la localización, las características y los obstáculos. Esta información es procesada por un grupo de ingenieros de software, que se aseguran que incluya todas las señales, pasos de cebra, semáforos , etc. Cuando los coches son conducidos autónomamente, se registran los posibles cambios, actualizando el mapa. Los investigadores se mostraron sorprendidos por la elevada frecuencia con que se producían cambios en las carreteras por las que circularon los vehículos.

Sin embargo, la conducción autónoma aún presenta ciertos problemas. De acuerdo a Sebastian Thrun, ingeniero de Google y director del Laboratorio de Inteligencia Artificial de Stanford, el equipo de diseño continúa trabajando en la interpretación de las indicaciones hechas manualmente por un guardia de tráfico.

# 3. FUNDAMENTOS TEÓRICOS

En el presente trabajo se desarrolla un algoritmo de detección de obstáculos mediante visión estéreo. La visión estéreo ha sido, y sigue siento, una de las áreas más ampliamente estudiadas en visión por computador. Se trata de una técnica que permite obtener información 3D a partir de pares de imágenes en dos dimensiones.

El algoritmo implementado en este proyecto calcula el mapa de disparidad y, a partir de éste, la *u-v disparity* utilizando como entrada las imágenes rectificadas proporcionadas por el sistema estéreo comercial Bumblebee de Pointgrey [7].

En este capítulo se realiza una breve descripción de los fundamentos teóricos que se han utilizado a lo largo del proyecto.

## 3.1. PRINCIPIOS ÓPTICOS

En el presente epígrafe se introducen los conceptos ópticos necesarios para el proceso de captación de imágenes.

### 3.1.1. MODELO DE LENTE FINA

En el proceso de captación de una imagen, los rayos inciden de forma perpendicular sobre una lente, como se observa en la imagen 3.1. Esta lente los concentra en un único punto, foco, situado a una distancia *f* (distancia focal) de la misma. La única trayectoria de la luz que no provoca ninguna distorsión, por tanto, es aquella que atraviesa la lente por su centro de forma perpendicular, recibiendo el nombre de eje óptico.

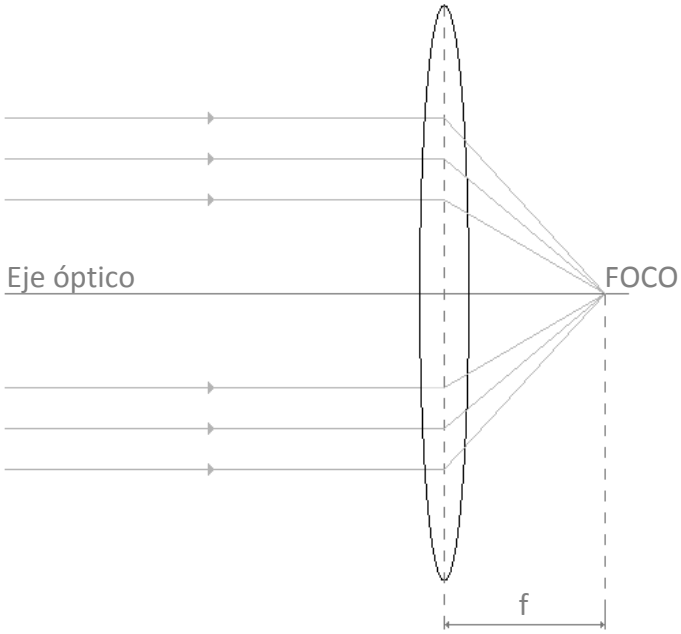



Figura 3.1 Modelo de lente fina

### 3.1.2. MODELO DE PIN-HOLE

El modelo *pin-hole* destaca por su sencillez, simplificando la óptica a un único punto situado a una distancia focal de la imagen, como se muestra en la figura 3.2. Usado en las primeras cámaras, se construye a partir de una caja cerrada con una apertura muy pequeña en un lado y la película fotográfica en el contrario.

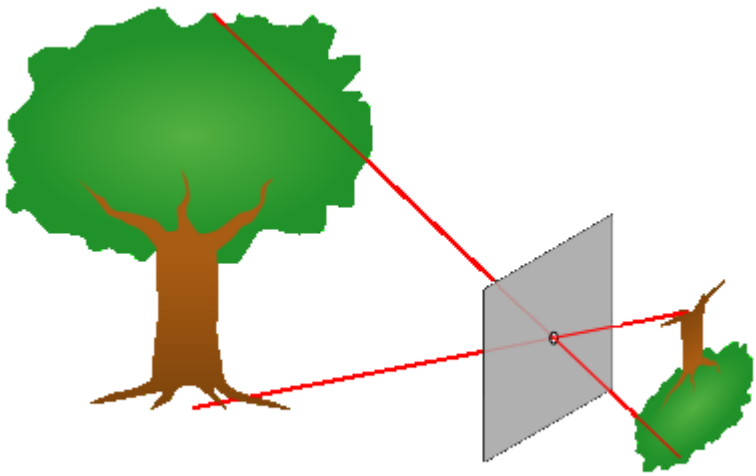


Figura 3.2 Modelo de cámara *pin-hole*.

Las ecuaciones que relacionan un punto P(W,U,V) del mundo con su proyección en la imagen (u,v) se obtienen por triangulación a partir de los parámetros indicados en la imagen 3.3:

$$u = \frac{f}{W} U \quad (1)$$

$$v = \frac{f}{W} V \quad (2)$$

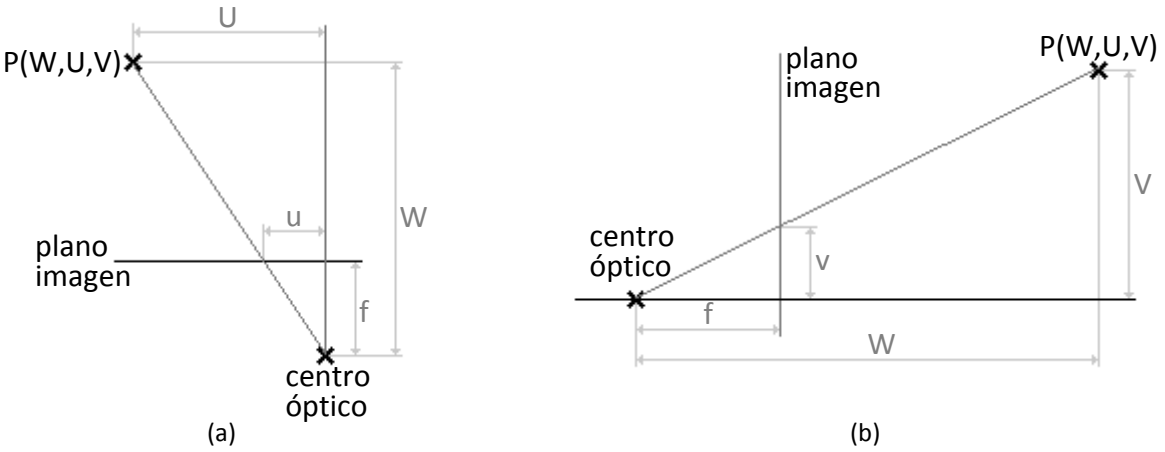



Figura 3.3 Parámetros del modelo *pin-hole*, (a) vista aérea, (b) vista lateral.

### 3.1.3. SISTEMA ESTÉREO IDEAL

Se considera un sistema estéreo ideal [14] aquel compuesto por dos cámaras cuyos ejes ópticos son paralelos y cuya distancia focal es la misma. Para este sistema, un mismo punto P(W,U,V) se proyecta en las imágenes en las posiciones $p_1(u_1,v_1)$ y $p_2(u_2,v_2)$. En general, las coordenadas de las dos proyecciones no coinciden, denominándose esta diferencia disparidad.

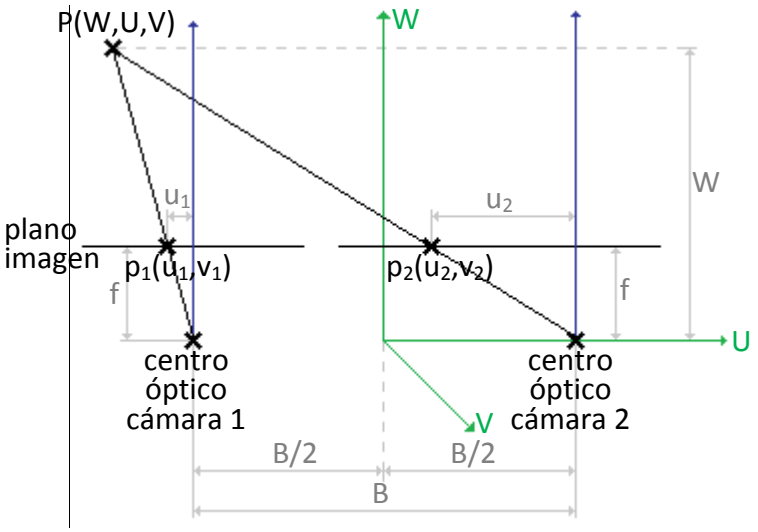



Figura 3.4 Modelo *pin-hole* para un sistema estéreo.

A partir de las ecuaciones del modelo *pin-hole*, se pueden plantear las ecuaciones del sistema estéreo con los parámetros definidos en la figura 3.4, donde B es la distancia entre las cámaras.

$$\frac{u_1}{f} = \frac{U - B/2}{W} \tag{3}$$

$$\frac{u_2}{f} = \frac{U + B/2}{W} \tag{4}$$

Despejando las ecuaciones (3) y (4), de acuerdo al desarrollo matemático siguiente, se llega a la expresión (10), que proporciona información sobre la profundidad (W) de un punto de la imagen en el mundo:

$$\frac{u_1.W}{f} + \frac{B}{2} = U \tag{5}$$

$$\frac{u_2.W}{f} - \frac{B}{2} = U \tag{6}$$

$$\frac{u_1.W}{f} + \frac{B}{2} = \frac{u_2.W}{f} - \frac{B}{2} \tag{7}$$

$$\frac{u_2.W}{f} - \frac{u_1.W}{f} = B \tag{8}$$

$$\frac{W}{f}(u_2 - u_1) = B \tag{9}$$

$$W = \frac{B.f}{(u_2 - u_1)} \tag{10}$$

### 3.1.4. GEOMETRÍA EPIPOLAR

La geometría epipolar, base del presente capítulo, comprende una serie de reglas que permiten relacionar los objetos tridimensionales con sus proyecciones en la imagen. Se trata de una extensión de los principios de la perspectiva aplicados a imágenes percibidas por diversos observadores.

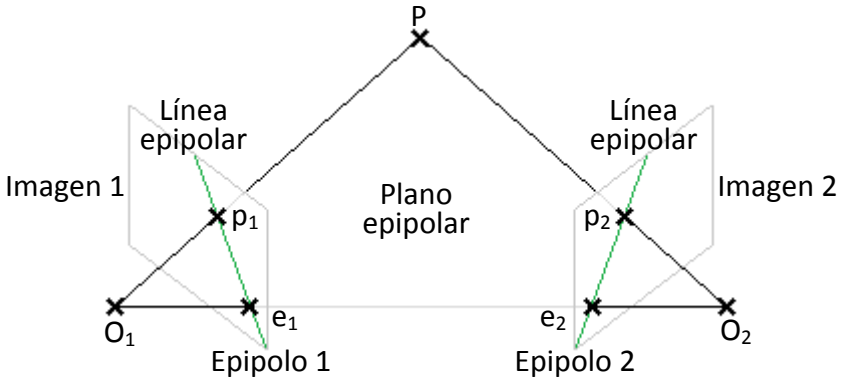



Figura 3.5 Geometría epipolar.

De acuerdo a la figura 3.5, se define el plano epipolar como aquel que contiene los dos centros ópticos, $O_1$ y $O_2$, y el punto P. Análogamente, se definen las líneas epipolares como la intersección del plano imagen con el plano epipolar. Los puntos de corte de la recta que une los centros ópticos de ambas cámaras con los planos de proyección son los epipolos ($e_1$ y $e_2$), de ahí el nombre de geometría epipolar.

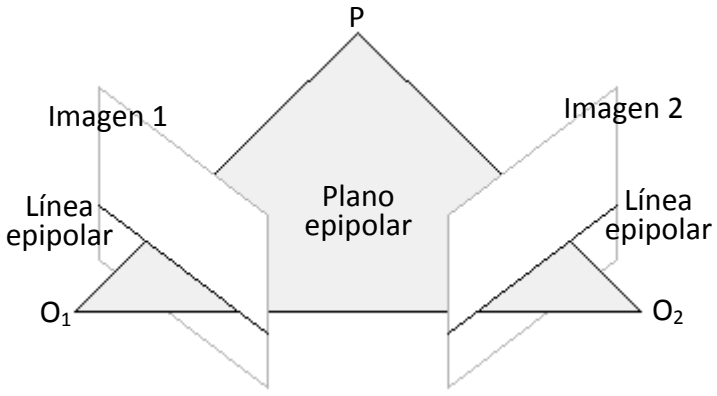



Figura 3.6 Geometría epipolar para una imagen normalizada.

En una imagen rectificada, las líneas epipolares son paralelas a las filas, es decir, pueden representarse como rectas horizontales como se observa en la imagen 3.6.

Esto se cumple cuando los dos ejes de las cámaras del sistema estéreo son paralelos entre sí. En las imágenes rectificada se cumple la restricción epipolar:

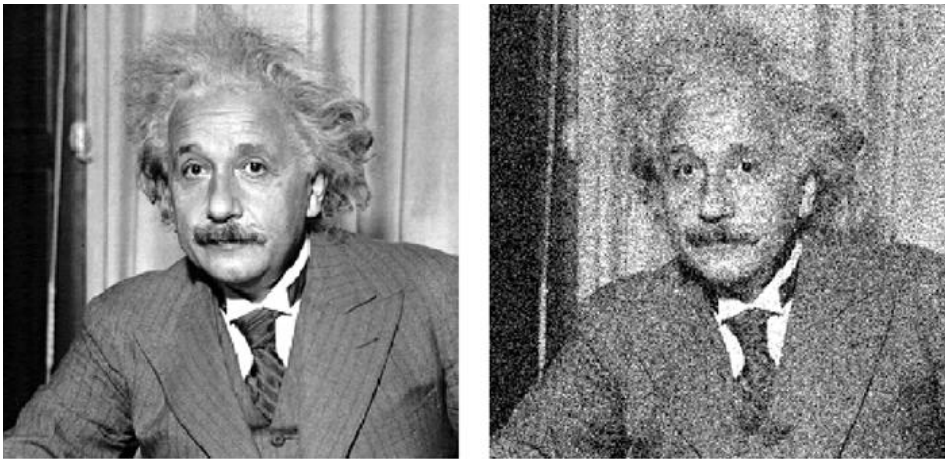
$$v_1 = v_2 \tag{11}$$

Lo que permite que el problema de *matching* entre las imágenes izquierda y derecha pase, de ser un problema 2D, a un problema 1D.

## 3.2. ELIMINACIÓN DEL RUIDO

Todas las imágenes tienen una cierta cantidad de ruido [10]. Las causas de esta distorsión pueden ser diversas, debiéndose habitualmente al sensor de la cámara y al medio físico de transmisión de la señal. Habitualmente, este ruido se manifiesta en píxeles aislados que toman niveles de gris diferentes al ideal. El ruido se puede clasificar en cuatro tipos:

### 3.2.1. TIPOS DE RUIDO



(a) (b)

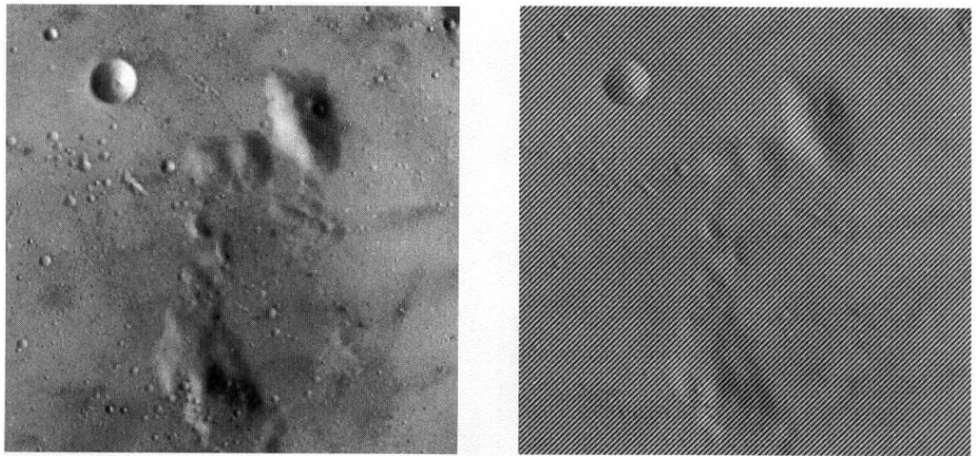
Figura 3.7 (a) Imagen original, (b) imagen afectada por ruido gaussiano.

RUIDO GAUSSIANO: Produce pequeñas variaciones en la imagen, tal como se observa en el par de fotografías anterior (3.7), de tal forma que el valor final del píxel es el ideal más un error (que puede describirse como una distribución normal o gaussiana). Habitualmente es causado por componentes electrónicos (sensores, digitalizadores, etc.).

(a) (b)

Figura 3.8 (a) Imagen original, (b) imagen afectada por ruido impulsional.

RUIDO IMPULSIONAL: Es conocido como “sal y pimienta” porque el nivel de gris de los píxeles afectados no tiene relación alguna con los píxeles vecinos, tomando valores máximos (sal) o mínimos (pimienta) del nivel de gris, como se observa en la figura 3.8.



(a) (b)

Figura 3.9 (a) Imagen original, (b) Imagen afectada por ruido frecuencial.

RUIDO FRECUENCIAL: La imagen final es la real más una interferencia de señal periódica (habitualmente una senoide), como se aprecia en la figura 3.9.

(a) (b)

Figura 3.10 (a) Imagen original, (b) imagen afectada por ruido multiplicativo.

RUIDO MULTIPLICATIVO: La señal obtenida es fruto de la multiplicación de dos señales. Esta naturaleza multiplicativa conlleva que las zonas más claras de la imagen sean afectadas por un elevado nivel de ruido (llegando a desbordar el píxel y, por tanto, adquiriendo tonos oscuros, como se observa en el cielo de la imagen 3.10), mientras que en las oscuras apenas se produce distorsión. Al depender el ruido de los niveles de intensidad de los píxeles afectados, no es fácil establecer un modelo estadístico del ruido estudiando la imagen afectada, lo que dificulta su eliminación [23].

### 3.2.2. FILTROS FRECUENCIALES

Antes de comenzar a trabajar con las imágenes recogidas y rectificadas por la cámara estéreo, se aplica un filtro que minimice el ruido presente en las mismas, reduciendo en la medida de lo posible los errores en la comparación de ambas imágenes.

#### FILTRO DE PASO BAJO

Al considerar el ruido como variaciones de importancia sobre los niveles de gris entre píxeles vecinos, se asume que le corresponden frecuencias altas. Por ello, para su eliminación, se pueden emplear filtros paso bajo como el que se presenta a continuación.

$$k.\begin{bmatrix} 1 & b & 1 \\ b & b^2 & b \\ 1 & b & 1 \end{bmatrix} \quad \text{con } k = (b^2 + 4.b + 4)^{-1} \qquad (12)$$

De esta forma se consigue reducir la ganancia a dichas frecuencias, consiguiendo que el nuevo valor del píxel sea proporcional al nivel de gris de los píxeles vecinos (12). Este tipo de filtros presentan el inconveniente de que, además de eliminar parte del ruido, también desdibujan los contornos, pudiendo llegar a eliminar objetos de pequeño tamaño.

**FILTRO DE GAUSS**

Otro tipo de filtros son aquellos que intentan imitar la forma de una gaussiana (13), donde *u* y *v* son las posiciones de los píxeles vecinos respecto del central.

$$G(u,v) = e^{-\frac{(u+v)^2}{2\sigma^2}} \quad (13)$$

Por ejemplo, para una desviación típica de σ=0,391, quedaría:

$$\frac{1}{32}.\begin{bmatrix}1 & 4 & 1\\ 4 & 12 & 4\\ 1 & 4 & 1\end{bmatrix} \quad (14)$$

Sin embargo, estos filtros presentan las mismas desventajas que los filtros paso bajo.

## 3.3. FILTROS ORIENTADOS A LA DETECCIÓN DE BORDES

Los bordes de los objetos son representados en una imagen como transiciones entre píxeles con niveles de gris significativamente distintos, pudiendo ser usada esta característica para la segmentación. La información de los bordes se localiza en las altas frecuencias, siendo necesario, por tanto, el uso de filtros de paso alto para su detección.

### 3.3.1. OPERADORES BASADOS EN LA PRIMERA DERIVADA (GRADIENTE)

Los bordes de una imagen, entendidos como transiciones entre niveles de gris, se pueden representar como una función escalón. Considerando que la derivada de una señal continua proporciona las variaciones locales con respecto a dicha variable (cuanto más rápida es la variación, mayor la derivada), ésta se puede usar para la detección

de bordes. Sin embargo, se ha descartado el uso de un operador basado en la primera derivada al presentar las siguientes desventajas [27]:

- Gran sensibilidad al ruido.
- Anchura de los bordes detectados de varios píxeles.
- Mala respuesta en bordes diagonales (el gradiente presenta una excesiva dependencia con respecto a la dirección de barrido, por ello, las aristas cuyas pendientes están próximas a la dirección de barrido no se detectan fácilmente).
- La debilidad del gradiente en los puntos esquina provoca la pérdida de puntos relevantes y marcado de junturas.

### 3.3.2. OPERADORES BASADOS EN LA SEGUNDA DERIVADA (LAPLACIANA)

Cuando la imagen presenta un cambio de intensidad en una dirección, genera un máximo en la primera derivada y, por tanto, un paso por cero de la segunda. Con los métodos basados en la segunda derivada se localizan los bordes como transiciones de positivo a negativo y viceversa.

Para analizar la imagen de forma multidireccional, se usan operadores independientes de la orientación. Habitualmente, los filtros usados para el cálculo de la laplaciana son:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad (15)$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad (16)$$

De las dos opciones expuestas anteriormente, (15) y (16), se ha elegido la segunda porque, al tener ceros en las esquinas, requiere cuatro operaciones menos por píxel.

Mediante la laplaciana es posible identificar bordes de un píxel de ancho, siendo el resultado independiente de la dirección del mismo. Sin embargo, al ser un filtro de

paso alto, la laplaciana es muy sensible al ruido por lo que, generalmente, no se aplica de forma aislada, sino en combinación con un filtro de paso bajo que elimine previamente el ruido en frecuencias altas.

### 3.3.3. OPERADOR DE MARR-HILDRETH

Un ejemplo de combinación de un filtro paso bajo y uno paso alto es la laplaciana de una gaussiana, construida como una convolución de la gaussiana con la laplaciana. Cuanto más estrecha sea esta laplaciana de la gaussiana, más bordes se obtendrán.

$$\frac{1}{32}.\begin{bmatrix} 0 & -1 & -4 & -1 & 0 \\ -1 & -4 & 2 & -4 & -1 \\ -4 & 2 & 32 & 2 & -4 \\ -1 & -4 & 2 & -4 & -1 \\ 0 & -1 & -4 & -1 & 0 \end{bmatrix} \quad (17)$$

Para el cálculo del operador de Marr-Hildreth (17), método basado en la laplaciana de la gaussiana, se ha partido de las matrices (14) y (16). De esta forma se consigue que el filtro final tenga cuatro valores nulos, simplificando los cálculos.

## 3.4. MAPA DE DISPARIDAD

El mapa denso de disparidad es una imagen donde la intensidad de cada píxel indica la profundidad de ese punto de la imagen en el mundo. El término disparidad fue introducido, en la literatura sobre visión humana, para describir las diferencias en la localización de un mismo rasgo entre la imagen percibida por el ojo izquierdo y la percibida por el derecho [33].

En visión por computador, calcular la disparidad es hallar la distancia que separa la proyección de un punto de la imagen izquierda con la proyección correspondiente en la imagen derecha. Este tipo de correspondencia estéreo es una de las áreas de visión por computador más estudiadas.

Con el objetivo de centrase únicamente en la interpretación de las imágenes, éstas se suponen tomadas por una cámara estéreo de geometría conocida (consultar capítulo 3.1). Además, cualquier algoritmo de visión realiza una serie de asunciones, explícita o implícitamente, sobre el mundo físico y el proceso de captación de las imágenes:

- En primer lugar, se parte del hecho de que para un punto del mundo sus proyecciones en ambas imágenes deben tener un nivel de gris idéntico. Para cumplir esta condición, es necesario asumir, entre otros, que el aspecto de las superficies que aparecen en la imagen no varía en función del punto de vista.

- Además, es habitual trabajar a partir de un par de imágenes rectificadas en las que se cumple la geometría epipolar, $y_{izda} = y_{dcha}$. Para poder realizar esta asunción es necesaria una etapa previa de rectificación.

El presente estudio se centra en el cálculo de la disparidad (horizontal). Se supone, por tanto, que los ejes ópticos de ambas cámaras son paralelos. En estas condiciones, la correspondencia entre un píxel $(u_i,v_i)$ de la imagen izquierda y uno $(u_d,v_d)$ de la imagen derecha viene dada por las siguientes expresiones:

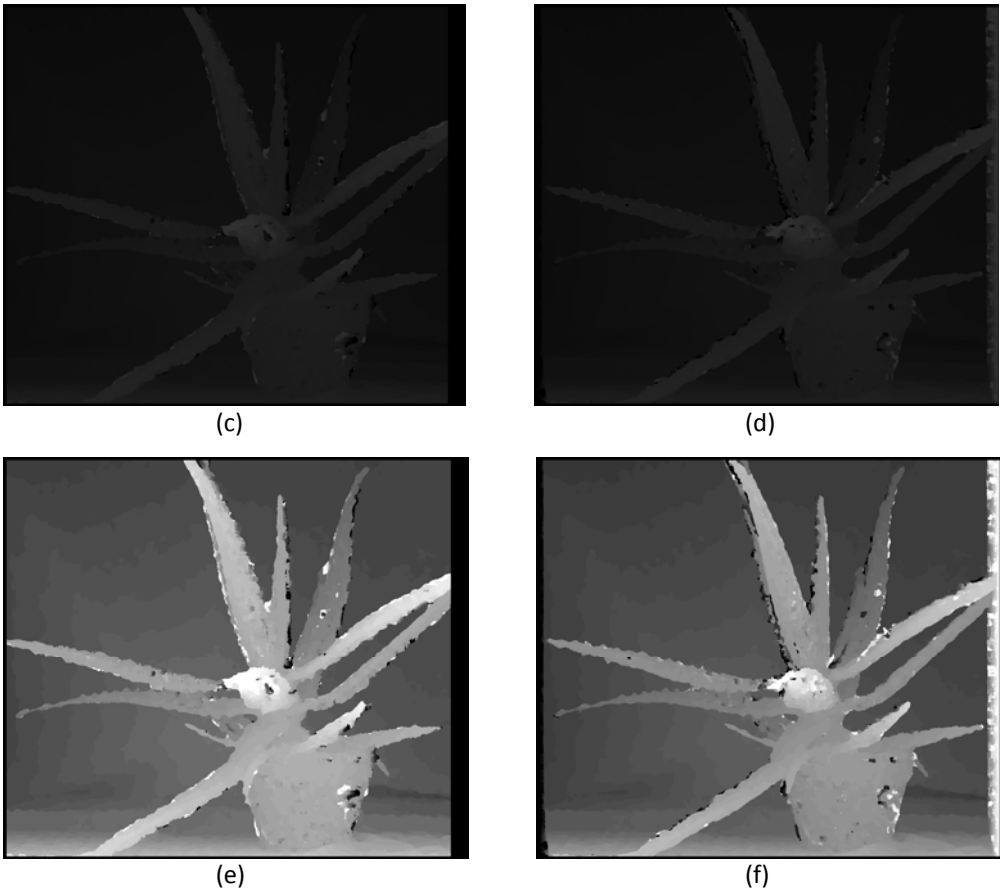
$$u_d = u_i + d(u_i,v_i) \quad (18)$$

$$v_d = v_i \quad (19)$$

donde $d(u_i,v_i)$ es la disparidad del punto $(u_i,v_i)$. El mapa de disparidad para este par de imágenes almacena en la coordenada $(u_i,v_i)$ el valor de $d(u_i,v_i)$.

(a)

(b)



(c) (d)

(e) (f)

Figura 3.11 (a), (b) Imágenes izquierda y derecha [20]; (c), (d) mapas de disparidad; (e), (f) mapas de disparidad equalizados[1].

El resultado del cálculo de la disparidad entre la imagen izquierda (a) y la derecha (b) se observa en la imagen (c). En la (d) se aprecia el mapa denso de disparidad de la imagen derecha con respecto a la izquierda. En ambos casos se han equalizado las imágenes resultantes, (e) y (f), para facilitar su interpretación.

En el algoritmo de cálculo del mapa denso de disparidad pueden diferenciarse cuatro etapas, siguiendo la taxonomía presentada por Scharstein y Szeliski en [33]:

- Cálculo de la función de coste.
- Agregación del coste.
- Cálculo de la disparidad.
- Post-procesamiento.

[1] En el presente, algoritmo el mapa de disparidad toma valores entre 0 y 30. Para facilitar el análisis visual de las imágenes, se han ajustado los niveles de gris entre 0 y 255.

### 3.4.1. CÁLCULO DE LA FUNCIÓN DE COSTE

Existen diversos métodos para evaluar el nivel de semejanza entre un par de píxeles. Los dos más habituales, basados en el coste, son:

- Diferencias cuadráticas de intensidad (*SD* ≡ *Squared Intensity Differences*) donde, como su propio nombre indica, se haya la diferencia cuadrática entre las intensidades. Cuanto menor es el valor resultante, mayor similitud existe entre los píxeles comparados.
- Diferencias absolutas de intensidad (*AD* ≡ *Absolute Intensity Differences*). En este caso se calcula la diferencia absoluta entre los dos valores de intensidad.

Tradicionalmente, se han usado también otros métodos, como la *normalized cross-correlation*, que se comporta de forma similar a la suma de diferencias cuadráticas (*SSD* ≡ *Sum of Squared Differences*), o la correspondencia binaria. Más recientemente, otros han sido propuestos, como la *truncated quadratics* [33] y la *contaminated Gaussians* [33], por su capacidad de limitar la influencia de correspondencias erróneas en posteriores etapas del algoritmo.

### 3.4.2. AGREGACIÓN DEL COSTE

Los métodos basados en ventanas (ver capítulo 3.4.3) trabajan a partir de grupos de píxeles. Se comparan los valores de intensidad de los píxeles que forman la región de soporte, agregándose los costes sumados o promediados. Estas regiones pueden ser de diversos tipos [2]:

- Ventana rectangular: Los algoritmos de ventana rectangular son los más sencillos. Se basan en la agregación de los costes de una ventana rectangular de tamaño definido alrededor del píxel de interés.

- Ventana adaptativa: Existe gran variabilidad entre los algoritmos que desarrollan este método, consistente en el uso de ventanas de diferentes formas, tamaños y posiciones, para obtener diversas estimaciones de la similitud entre píxeles. La disparidad

se determina eligiendo la ventana que ha obtenido menor coste o mediante la ponderación de los resultados obtenidos.

- Pesos adaptativos: Este tipo de algoritmos es una evolución de los anteriores, otorgando mayor peso a los píxeles de la ventana que tienen mayor probabilidad de pertenecer al mismo objeto que el píxel central. Para calcular dicha probabilidad se utilizan varios criterios, como la lejanía o la diferencia de color respecto al píxel central.

- Difusión iterativa: En los algoritmos de difusión, en lugar de agregar los costes de una ventana circundante al píxel de interés, se difunde el coste a los píxeles vecinos durante varias iteraciones, siendo necesario establecer una condición de parada para dicha difusión.

### 3.4.3. CÁLCULO DE LA DISPARIDAD

- Métodos locales: Los métodos locales enfatizan las etapas de correspondencia y agregación. El cálculo de la disparidad, por contra, resulta trivial: la posición con menor coste asociado marca su valor (*Winner Take All* ≡ *WTA*). La limitación que presentan este tipo de procesos es el hecho de que sólo se tenga en cuenta una única correspondencia.

- Optimizaciones globales: Gran parte del trabajo de los métodos globales se desarrolla durante esta fase, saltándose frecuentemente la agregación. La mayoría están formulados con la premisa de minimizar la energía. Para ello, se busca una función de disparidad *d* que minimice esta energía global. Posteriormente, se usan diversos algoritmos para hallar el mínimo de *d*, que constituye la disparidad buscada.

Recientemente, se han propuesto los métodos *max-flow* [33] y *graph-cut* [33] para resolver algunas clases de problemas de optimización global, obteniendo buenos resultados de forma más eficiente.

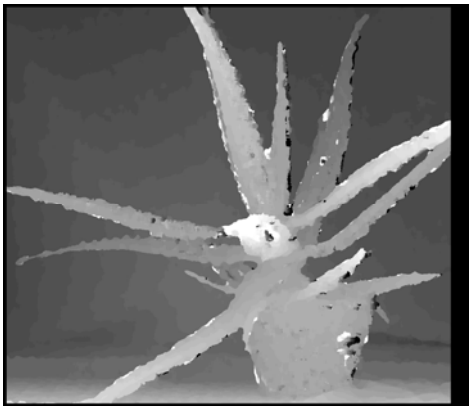
- Programación dinámica: Se trata de un tipo de optimización global donde, a través de una programación dinámica, se localiza el mínimo global.

- Algoritmos cooperativos: Inspirados por los modelos humanos de visión estéreo, usan operaciones no-lineales cuyo resultado final se asemeja a los algoritmos de optimización globales.

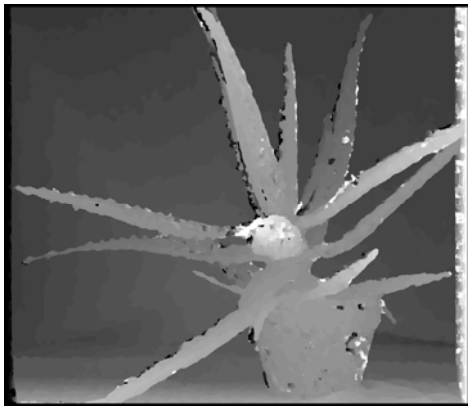
### 3.4.4. POST-PROCESAMIENTO

Para corregir posibles discontinuidades en el mapa de disparidad obtenido, muchos algoritmos incluyen una etapa final de refinamiento tras la inicial de correspondencia discreta.

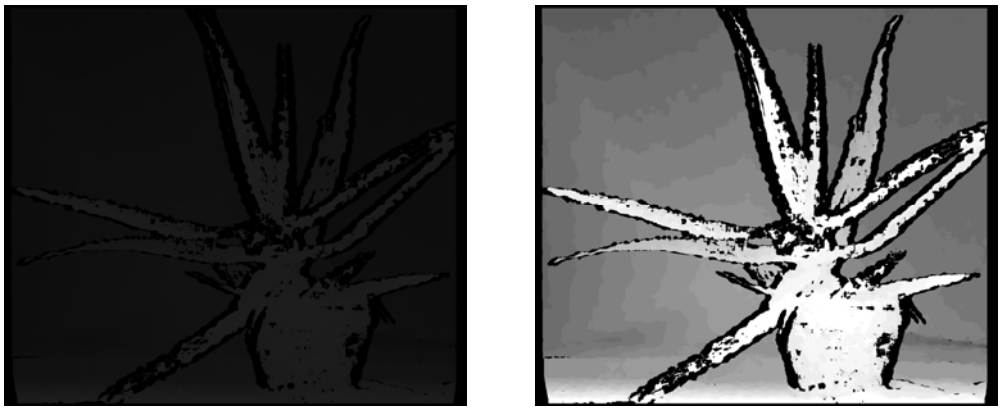
El refinamiento de la disparidad puede ser implementado mediante diversos procedimientos, incluyendo el gradiente descendiente iterativo [33] y el ajuste a una curva de los niveles de la disparidad [33]. De esta forma se aumenta la resolución del algoritmo estéreo, con un pequeño incremento del coste computacional. Sin embargo, para que el proceso sea efectivo, las intensidades deben variar suavemente y las regiones sobre las que se aplica, pertenecer a la misma superficie.



(a)



(b)



(c) (d)

Figura 3.12 (a) Disparidad de la imagen izquierda con respecto a la derecha (imagen equalizada), (b) disparidad de la imagen derecha con respecto a la izquierda (imagen equalizada), (c) disparidad resultante tras el *cross-checking*, (d) disparidad resultante tras el *cross-checking* (imagen equalizada).

Además, existen otros métodos de post-procesamiento de las disparidades calculadas. Las áreas ocluidas pueden ser detectadas mediante el *cross-checking* (d). Consistente en comparar el mapa de disparidad de la imagen izquierda con respecto a la derecha (a) con el de la imagen derecha con respecto a la izquierda (b), marcando como píxeles ocluidos aquéllos cuyo valor de disparidad sea diferente.

Una vez localizadas, las zonas ocluidas o sin disparidad se pueden completar mediante una interpolación con las disparidades más cercanas, o asignándolas un valor de disparidad determinado (por ejemplo, con la imagen izquierda como imagen referencia, con el valor de disparidad más cercano por la izquierda). También es posible aplicar un filtro que elimine el ruido y conserve los bordes, como los bilaterales y de mediana, para eliminar correspondencias erróneas y completar estas zonas ocluidas.

## 3.5. U-V DISPARITY

Las imágenes *u-v disparity* [22] se construyen a partir del mapa de disparidad. Ambas contienen sus histogramas laterales, columna a columna en el caso de la *u_disparity* y por filas en el de la *v_disparity*.

Los obstáculos situados perpendicularmente aparecen representados en la *u_disparity* por líneas horizontales, siendo su intensidad la altura de los mismos medi-

da en píxeles, como se observa en la imagen 3.13. En la *v_disparity*, por contra, aparecen como líneas verticales, su intensidad, en este caso, corresponde a su anchura medida en píxeles. En la *v_disparity*, además, aparece una línea oblicua marcando el *road profile* (perfil del suelo).

Figura 3.13 Mapa de disparidad (imagen equalizada), (derecha) *v_disparity*, (abajo) *u_disparity*.

## 3.6. TRANSFORMADA DE HOUGH

La transformada de Hough es un algoritmo de reconocimiento de patrones [14]. En el presente proyecto se usa la transformada de Hough para rectas con el objetivo de calcular el *road profile* a partir de la *v_disparity*.

Para evitar singularidades, tales como rectas de pendiente infinita, se usa una representación paramétrica de la ecuación de la recta definida por:

$$x \cdot \cos\theta + y \cdot \mathrm{sen}\theta = \rho \qquad (20)$$

donde, como se observa en la figura 3.14, ρ representa la distancia entre la recta y el origen de coordenadas y θ, el ángulo formado ρ y el eje de abscisas.

El método de la transformada de Hough tiene como inconveniente su elevado coste computacional. Se va a usar la implementación probabilística de dicha transformada contenida en las librerías OpenCv. Este método calcula todas las rectas que pasan por cada punto de la imagen, dando como resultado aquella que más se haya repetido, que es la que contiene mayor número de píxeles.

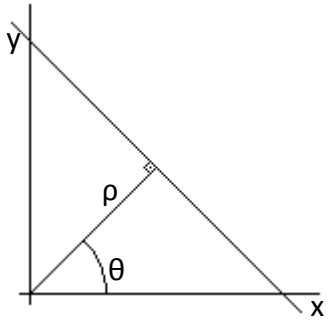



Figura 3.14 Representación gráfica de los parámetros paramétricos y cartesianos.

## 3.7. BLOB ANALYSIS

El *Blob Analysis* es un proceso que estudia los grupos de píxeles que aparecen en la imagen. Para ello, se parte de una imagen binarizada. Este análisis detecta las regiones en la imagen más claras/oscuras que los alrededores, denominadas Blobs (*Binary Large Object*) [21].

Una vez localizadas las áreas de píxeles contiguos con el mismo estado lógico, su estudio proporciona información diversa: tamaño, perímetro, número de blobs, localización, etc.

Estas herramienta están ampliamente implantada en el ámbito de la inspección de procesos automatizados tales como:

- Detección de defectos en obleas de silicio.
- Detección de defectos de soldadura en placas electrónicas.
- Detección de objetos en aplicaciones con gran variabilidad en forma y/o orientación, como la industria farmacéutica.

Frente a los sistemas de segmentación basados en la detección de bordes, el *Blob Analysis* es más rápido. Por contra, cuando las condiciones del entorno varían, más específicamente, cuando no hay luz suficiente, el análisis puede no ser suficientemente robusto, debido a que esta técnica depende de poder diferenciar los objetos que aparecen en la imagen respecto del fondo de la misma.

El análisis de Blobs será usado para determinar las ROIs (*ROI* ≡ *Region of Interest*) alrededor de cada uno de los obstáculos delante del vehículo.

# 4. DESARROLLO DEL ALGORITMO

El presente capítulo tiene como objeto la descripción del algoritmo de detección y localización de obstáculos en entornos urbanos mediante visión estéreo. Este proceso se compone de tres etapas:

- Construcción del mapa denso de disparidad.
- Detección de los obstáculos en la imagen.
- Localización de los obstáculos respecto del vehículo.

A continuación se detalla cada una de ellas, indicando aquellas características o parámetros de funcionamiento, como pueden ser el tamaño de las imágenes, la disparidad máxima, el tamaño de ventana, etc. que influyen en el tiempo de computo para su posterior evaluación.

## 4.1. MAPA DE DISPARIDAD

El cálculo del mapa denso de disparidad es el punto crítico del algoritmo desde un punto de vista computacional. Para su desarrollo, se parte del proceso básico de comparación de dos imágenes píxel a píxel (*matching problem*). En las siguientes versiones el proceso gana en complejidad con objeto de disminuir el coste computacional. Entre la primera versión y la última se ha conseguido reducir el tiempo de cómputo en más de un 1.000% (consultar capítulo 5.1).

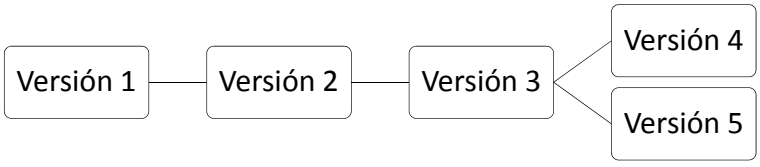



Figura 4.1 Evolución del algoritmo de cálculo de la disparidad.

La evolución del algoritmo no ha sido lineal, como puede verse en la figura 4.1, sino que se han desarrollado dos esquemas distintos de forma paralela. A lo largo del

presente capítulo se detalla el funcionamiento de cada una de las implementaciones. La opción óptima, la versión 4, se analiza en el siguiente capítulo.

### 4.1.1. VERSIÓN 1

**CÁLCULO DEL MAPA DE DISPARIDAD**

Como se comentó anteriormente, el mapa denso de disparidad es una imagen cuyos niveles de gris indican la distancia horizontal, en píxeles, entre las proyecciones de un punto del mundo en cada uno de los planos de imagen de cada cámara.

A modo de simplificación, se supondrán las cámaras calibradas y las imágenes rectificadas, de modo que la búsqueda se reduce a una línea horizontal (línea epipolar). Por tanto, la coordenada *y* es común a ambas imágenes (Capítulo 3.1.4).

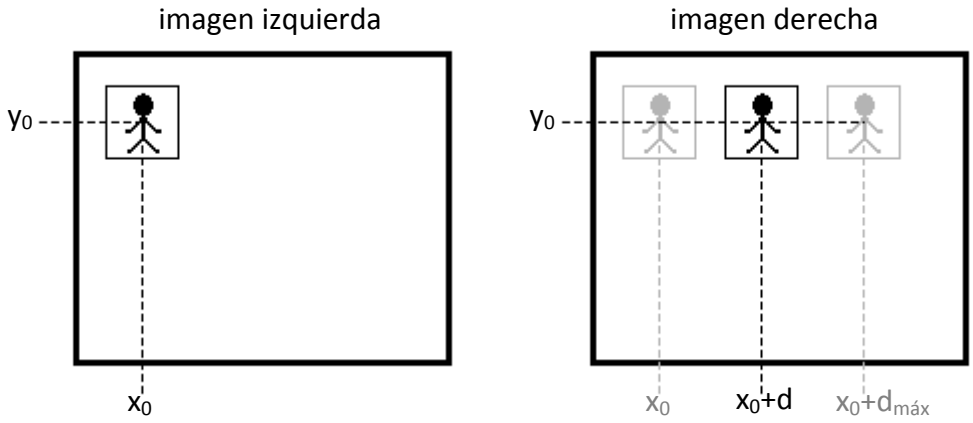



Figura 4.2 Proceso de comparación entre las dos imágenes estéreo.

Tomando una ventana de la imagen izquierda como referencia se busca, en la imagen derecha, la ventana que ofrezca mayor similitud. Para una ventana de la imagen izquierda centrada en el píxel $(x_0,y_0)_{izda}$, la búsqueda comienza en la misma posición de la imagen derecha $(x_0,y_0)_{dcha}$ prolongándose hasta $(x_0+d_{máx},y_o)_{dcha}$, donde $d_{máx}$ es la disparidad máxima (ver figura 4.2).

Para analizar la similitud entre dos píxeles se ha elegido como función de coste, en base a su mayor facilidad de implementación, el método de las diferencias absolutas de las intensidades. El valor resultante se nombra, a lo largo del presente ensayo, co-

mo AD (*Absolute Difference*). Por su parte, la suma de los AD de todos los píxeles que componen la ventana, correspondiente a la etapa de agregación de coste, se nombra, de forma indistinta, como coste o SAD (*Sum of Absolute Differences*).

En el vector coste se almacenan los valores del coste de comparar cada par de ventanas $(x_0,y_0)_{izda}$ con $(x_0,y_0)_{dcha}$ y posteriores hasta $(x_0+d_{máx},y_o)_{dcha}$. Siguiendo el cálculo de la disparidad mediante el método local WTA, una vez hallados todos los valores de este vector coste, se comparan en busca del menor, cuya posición marca la disparidad del píxel estudiado, $(x_0, y_0)$.

En este algoritmo, para el cálculo de la SAD entre dos ventanas se requiere comparar uno a uno todos los píxeles que las componen. Esto se hace mediante un bucle de la forma:

```
desde j=0 hasta j=n
   desde i=0 hasta i=n
      coste=coste+abs(imagen_izda[j,i]–imagen_dcha[j,i])
   fin desde
fin desde
```

Por tanto, para el cálculo de cada SAD se requieren $4.n^2$ accesos a memoria, donde n es el tamaño del lado de la ventana, y $2.n^2$ operaciones matemáticas. Notar que, por simplicidad, el valor absoluto no se computa como operación matemática.

### 4.1.2. VERSIÓN 2

#### CÁLCULO DEL MAPA DE DISPARIDAD A PARTIR DEL PRIMER ELEMENTO DE CADA COLUMNA

En esta primera evolución se busca reducir el coste computacional. Para ello, la coste de agregación de la primera ventana de cada columna se calcula siguiendo el patrón expuesto en el apartado anterior (esto es, comparando todas las parejas de píxeles que forman la ventana). Este resultado de comparar una ventana de la imagen izquierda centrada en $(x_0,y_0)_{izda}$ con la análoga de la imagen derecha centrada en $(x_0,y_0)_{dcha}$ y posteriores, hasta la centrada en $(x_0+d_{máx},y_0)_{dcha}$, se almacena en la posición *d* el vector coste, donde la disparidad *d* toma valores entre 0 y $d_{máx}$.

Para el cálculo de las demás ventanas de cada columna, se aprovecha el solapamiento que se produce entre dos ventanas consecutivas (superior e inferior). A partir de los valores almacenados en el vector coste, correspondientes a la ventana ($x_0$, $y_0$), se calculan los de la ($x_0$,$y_0$+1) restando los valores de las AD de los píxeles de la fila ($y_0$-w), con $w = \frac{n-1}{2}$, que únicamente pertenece a la anterior ventana, y sumando los de la fila ($y_0$+1+w), privativa de la nueva ventana, como se observa en la figura 4.3.

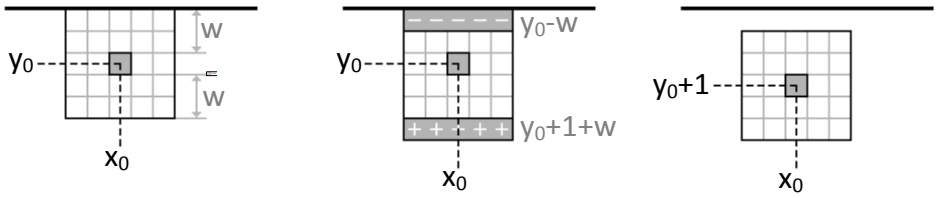



Figura 4.3 Esquema del cálculo del coste de agregación para una ventana centrada en el píxel ($x_0$,$y_0$+1) utilizando el coste de la ventana centrada en ($x_0$,$y_0$).

En el esquema previo (figura 4.3) se asume que la ventana ($x_0$, $y_0$) pertenece al borde de la imagen, sin embargo, el proceso es aplicable a cualquier par de ventanas que pertenezcan a la misma columna y cuya coordenada *y* sea correlativa.

En este algoritmo, para el cálculo del coste entre dos ventanas que no pertenezcan al borde superior de la imagen, se requieren 6.n accesos a memoria y 4.n operaciones, donde n es el tamaño del lado de la ventana:

```
desde i=0 hasta i=n
  coste=coste +abs(imagen_izda[i,y0+1+w]–imagen_dcha[i,y0+1+w ])
              -abs(imagen_izda[i,y0]    –imagen_dcha[i,y0]     )
fin desde
```

Para las ventanas pertenecientes al borde superior se requieren $4.n^2$ accesos a memoria y $2.n^2$ operaciones, como se ha justificado en la versión anterior. Sin embargo, el coste de esta primera fila de ventanas tiene un peso despreciable frente al del resto de la imagen.

### 4.1.3. VERSIÓN 3

**CÁLCULO DE LAS DISPARIDADES IZQUIERDA Y DERECHA DE FORMA SIMULTÁNEA**

Hasta ahora se ha trabajado en el desarrollo de un algoritmo de cálculo de la disparidad de la imagen izquierda con respecto a la derecha. Sin embargo, para una etapa posterior de *cross-checking*, se necesita también el mapa de disparidad de la imagen derecha con respecto a la izquierda. A continuación se detalla cómo es posible obtener el coste de agregación para el cálculo del mapa de disparidad de la imagen derecha respecto de la izquierda, reutilizando los resultados del mapa de disparidad previo [13].

En la figura 4.4 se observa el detalle de diversas ventanas de las matrices izquierda y derecha. En esta imagen, aunque las ventanas se suponen centradas en píxeles consecutivos, no se refleja el solapamiento con el objeto de facilitar su interpretación.

En la siguiente figura, la 4.5 (a), se indican los valores que toma el vector coste para una posición del mapa de disparidad de la imagen izquierda con respecto a la derecha y diversos vectores coste del mapa de disparidad de la imagen derecha con respecto a la izquierda (b). Se han sombreado las posiciones cuyos valores de agregación de coste se repiten en la construcción de ambos mapas. Iterando este razonamiento, se concluye que todos los valores necesarios para el cálculo del vector de coste para una posición $(x_0,y_0)_{dcha}$ en la imagen derecha han sido previamente obtenidos durante el proceso de cálculo del mapa de disparidad izquierdo. La correspondencia entre los costes de agregación necesarios para el cálculo de las disparidades izquierda y derecha se puede expresar como:

$$C_{izdo}(x_0, y_0)_{d=di} = C_{dcho}(x_0+d_i, y_0)_{d=di} \qquad (21)$$

Donde $C_{izdo}(x_0,y_0)_{d=di}$ es el valor del coste de agregación de la posición $d_i$ del vector coste para las coordenadas $(x_0,y_0)$ del mapa de disparidad de la imagen izquierda con respecto a la derecha. De forma análoga, $C_{dcho}(x_0+d_i,y_0)_{d=di}$ es el valor del coste de agregación de la posición $d_i$ del vector coste para las coordenadas $(x_0+d_i,y_0)$ del mapa de disparidad de la imagen derecha con respecto a la izquierda.


Figura 4.4 (a) Detalle de la imagen izquierda, donde se muestran diversas ventanas (nombradas como A,B,C...); (b) detalle de la imagen derecha, donde se muestran diversas ventanas (a,b,c...).

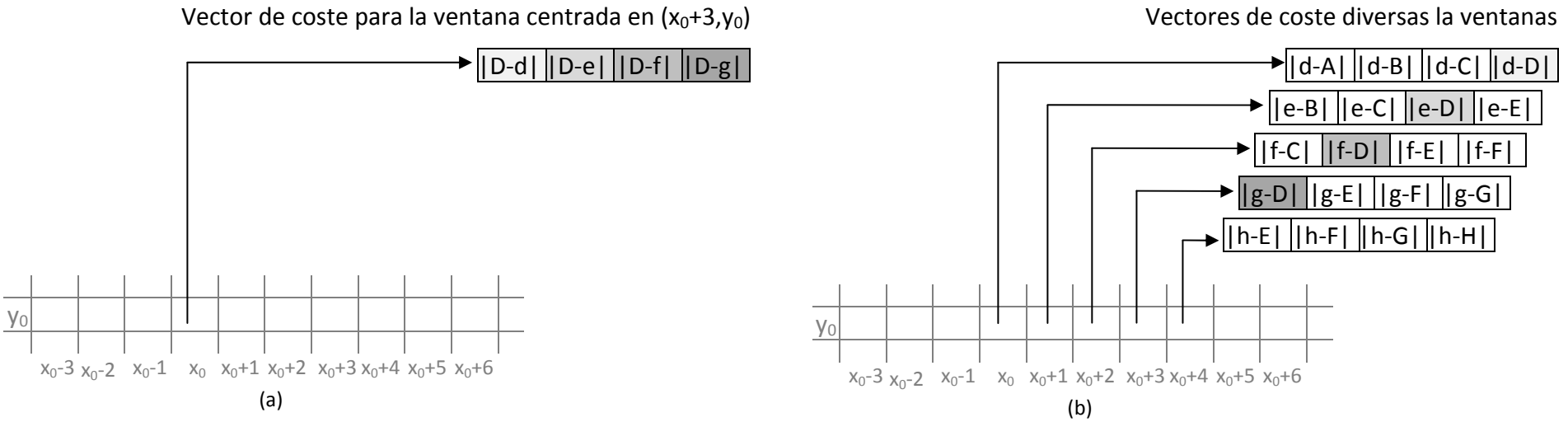



Figura 4.5 (a) Valores constitutivos del vector coste para una posición del mapa de disparidad de la imagen izquierda con respecto a la derecha, en función de la nomenclatura indicada en la figura 4.4, para una $d_{máx}$=3; (b) valores constitutivos de los vectores coste para diversas posiciones de la disparidad de la imagen derecha con respecto a la izquierda, con los valores coincidentes con el coste de la disparidad de la imagen izquierda con respecto a la derecha sombreados.

Durante el cálculo de la disparidad izquierda (no varía respecto a la Versión 2), los diversos costes resultantes de comparar una ventana centrada en $(x_0,y_0)_{izda}$ de la imagen izquierda con la correspondiente $(x_0,y_0)_{dcha}$ y sucesivas, hasta $(x_0+d_{máx},y_0)_{dcha}$, de la imagen derecha se almacenan en el vector coste, de tamaño $d_{máx}$. En el caso de la disparidad derecha, al calcularse los costes de agregación de forma desordenada, se requiere una matriz de las mismas dimensiones que las imágenes para almacenar el valor del menor coste encontrado para cada píxel. Al buscarse el coste de agregación (SAD) mínimo, que indique la mayor coincidencia entre ventanas, la matriz coste derecha se inicializa con el mayor valor posible de dicho coste. Esta metodología para determinar el valor de disparidad corresponde al método local (WTA) (Capítulo 3.4.3).

### 4.1.4. VERSIÓN 4

**CÁLCULO DEL MAPA DE DISPARIDAD MEDIANTE LA DESCOMPOSICIÓN DE LA VENTANA EN FILAS**

A partir de la Versión 3, manteniendo el cálculo simultáneo de los dos mapas de disparidad, se busca optimizar el algoritmo de obtención del coste de agregación, para lo que se ha dividido el proceso en cuatro partes para facilitar su exposición:

Columna izquierda · ventana superior

```
desde j=0 hasta j=n
   desde i=0 hasta i=n
      coste_fila[j]=coste_fila[j]+abs(im_izda[i,j]-im_dcha[i,j])
   fin desde
   coste=coste+coste_fila[j]
fin desde
```

Este algoritmo comienza calculando el valor del coste de agregación para la ventana superior izquierda. Para esta primera ventana de lado n es necesario hallar las AD de todos los píxeles que la forman. Sin embargo, estas AD calculadas no se almacenan en el vector coste, sino que el coste de agregación de cada una de las filas que conforman la ventana se almacena en una nueva matriz llamada coste_fila. Una vez calculado el coste_fila, se suma a los anteriores para obtener el coste de la ventana. En este primer paso, se ha introducido una etapa intermedia, el almacenamiento de los costes por filas, no registrándose más variaciones con respecto a algoritmos previos.

Columna izquierda · demás ventanas

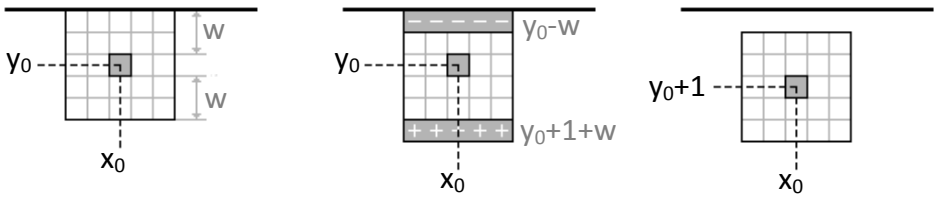



Figura 4.6 Solapamiento entre dos ventanas con la coordenada *x* correlativa.

```
desde i=0 hasta i=n
  coste_fila[y0+1+w]=coste_fila[y0+1+w]+abs(im_izda[i]-im_dcha[i])
fin desde
coste=coste+coste_fila[y0+1+w]-coste_fila[y-w]
```

Para el cálculo del resto de elementos de la primera columna se aprovecha el solapamiento entre ventanas con la coordenada *y* correlativa, esquematizado en la figura 4.6. De acuerdo al código previo, simplemente se calcula el valor de la nueva fila mediante un bucle, almacenándolo en la matriz coste_fila ($y_0$+1+w). A continuación, para actualizar el valor del coste de la ventana, se le suma la nueva fila ($y_0$+1+w), privativa de la ventana inferior, restándole la fila superior de la ventana anterior ($y_0$-w). En este caso se aprecia una mejoría con respecto a la versión anterior (versión 2), donde era necesario realizar el cálculo de ambas filas enteras.

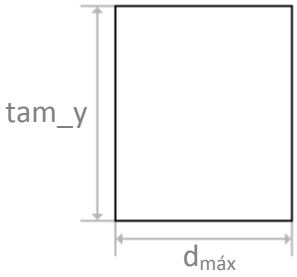



Figura 4.7 Dimensiones de la matriz coste_fila, donde tam_y es el tamaño vertical de las imágenes de partida y $d_{máx}$ el valor máximo de la disparidad.

El objetivo es almacenar para cada fila, en la matriz coste_fila (figura 4.7), los valores de los costes de agregación correspondientes a cada valor posible de la disparidad. De forma que, en la posición ($x_0$,$y_0$) de la matriz coste_fila se conserva el coste de agregación resultante de comparar los píxeles entre las posiciones ($x_0$-w,$y_0$) y ($x_0$+w,$y_0$) de la imagen izquierda con los que ocupan las mismas posiciones en la imagen derecha. En sentido de las abscisas, en la posición ($x_0$+d,$y_0$) se almacena el coste de compa-

rar las posiciones entre la ($x_0$-w,$y_0$) y la ($x_0$ +w,$y_0$) de la imagen izquierda con las comprendidas entre ($x_0$+d-w,$y_0$) y ($x_0$+d +w,$y_0$) de la imagen derecha. En el sentido de las ordenadas, en ($x_0$,$y_0$+k) se almacena el coste de agregación de la fila resultante de comparar los píxeles entre ($x_0$-w,$y_0$+k) y ($x_0$+w,$y_0$+k) de la imagen izquierda con los que ocupan idénticas posiciones en la imagen derecha.

Cuando se termina el cálculo de los costes de agregación para cada valor de disparidad de la primera columna, se obtiene la matriz coste_fila completa.

Demás columnas · ventana superior

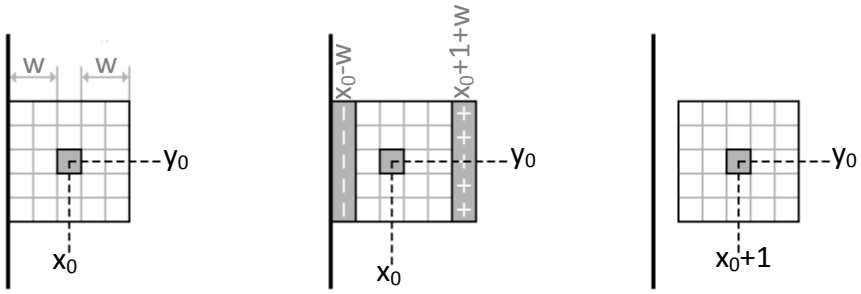



Figura 4.8 Solapamiento lateral entre dos ventanas con la coordenada *x* correlativa.

Como se indica en el esquema de la figura 4.8, es posible aprovechar también el solapamiento lateral (eje *x*) entre ventanas centradas en coordenadas *x* correlativas. Para la primera ventana de cada columna, es necesario actualizar los valores almacenados en coste_fila para cada una de las filas que constituyen dicha ventana:

```
desde j=0 hasta j=n
   coste_fila[j]=coste_fila[j]+abs(im_izda[x0+1+w]-im_dcha[x0+1+w])
                              -abs(im_izda[x0-w  ]-im_dcha[x0-w  ])
   coste=coste+coste_fila[j]
fin desde
```

De acuerdo al bucle anterior y al esquema de la figura 4.8, el algoritmo recorre la columna lateral de la ventana en sentido descendente. Para cada fila $y_i$, se actualiza el valor almacenado en coste_fila, eliminando la aportación del píxel ($x_0$-w,$y_i$), que únicamente pertenece a la anterior ventana, y añadiéndole la del píxel ($x_0$+1+w,$y_i$), privativo de la nueva. Los valores recién actualizados de las filas se almacenan, a medida que se van calculando, en el vector coste. A diferencia de la versión previa, al aprove-

char el solapamiento lateral se evita tener que agregar todas las AD de los píxeles que forman la ventana ($n^2$), siendo necesario calcular únicamente 2.n.

Demás columnas · demás ventanas

El resto de las ventanas de la imagen se pueden calcular haciendo uso de ambos solapamientos. Para una ventana que no pertenece a ningún borde de la imagen, el proceso de cálculo del coste de agregación:

```
coste_fila[y0+1+w]=coste_fila[y0+1+w]
                  +abs(im_izda[x0+1+w,y0+1+w ]-im_dcha[x0+1+w,y0+1+w])
                  -abs(im_izda[x0-w,y0+1+w   ]-im_dcha[x0-w,y0+1+w  ])
coste=coste+coste_fila[y0+1+w]-coste_fila[x0-w]
```

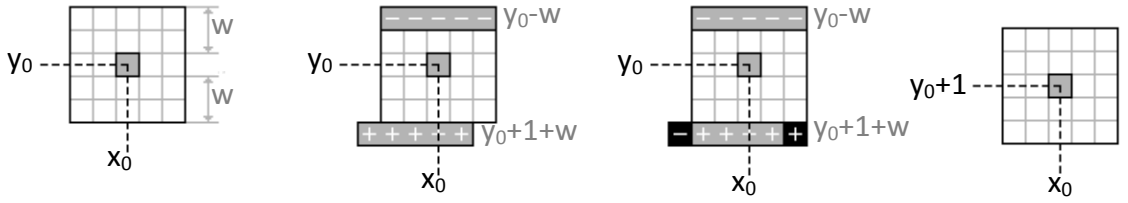



Figura 4.9 Esquema del cálculo del coste de agregación para la ventana centrada en ($x_0$,$y_0$+1).

Como se observa en la figura 4.9, para cualquier ventana de la imagen que no pertenezca a un borde, se puede calcular el nuevo coste a partir del de la ventana superior. Para ello, como en casos anteriores, se le resta el valor de la fila superior ($y_0$-w), que sólo pertenece a la ventana precedente, sumándole el de la fila inferior ($y_0$+1+w). Sin embargo, en este caso, el valor del coste de la fila superior ($y_0$-w) ya ha sido calculado con anterioridad, estando almacenado en la matriz coste_fila. En el caso de la fila inferior ($y_0$+1+w), hay que actualizarlo a la columna sobre la que se está trabajando, centrada en ($x_0$-1) puesto que el valor guardado corresponde a la columna anterior centrada ($x_0$). Para ello basta con quitarle la contribución del píxel ($x_0$-1-w,$y_0$+1+w) y sumarle la contribución del ($x_0$+w,$y_0$+1+w).

Por tanto, el coste computacional de calcular la disparidad entre un par de ventanas no pertenecientes ni a la primera fila ni a la primera columna de la imagen (la influencia sobre el tiempo de cómputo del resto de las ventanas se considera despreciable) es de diez accesos a memoria y seis operaciones matemáticas. Con esta nueva

implementación se consigue además hacer independiente el tiempo de cómputo del mapa de denso de disparidad del tamaño de la ventana (ver gráfica de resultados en el capítulo 5).

Como última mejora del algoritmo de obtención de la disparidad, se calculan todas las AD para cada uno de los posibles valores de disparidad al principio, en lugar de calcularlas a medida que se usan, evitándose de este modo repetir el cálculo:

```
desde j=0 hasta j= am_y
   desde i =0 hasta i=tam_x
      desde k=0 hasta k=d_máx
         AD[k,j,i]=AD[k,j,i]+abs(im_izda[k,j,i]-im_dcha[k,j,i])
      fin desde
   fin desde
fin desde
```

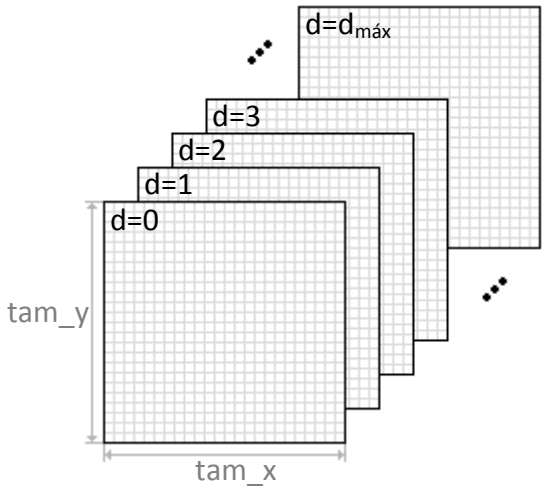



Figura 4.10 Dimensiones de la matriz AD, donde tam_y es el tamaño vertical de las imágenes de partida y tam_x el horizontal.

Las AD se almacenan en una matriz de tamaño tam_x * tam_y* $d_{máx}$ como se observa en la imagen 4.10. La matriz contiene $d_{máx}$ submatrices, en cada una de las cuales se conservan las AD de todos los píxeles la imagen para cada $d_i$ posible.

```
coste_fila[y0+1+w]=coste_fila[y0+1+w]+AD[x0+1+w,y0+1+w]-AD[x0-w,y0+1+w]]
coste=coste+coste_fila[y0+1+w]-coste_fila[x0-w];
```

De esta forma, los costes computacionales de calcular la disparidad entre un par de ventanas no pertenecientes ni a la primera fila ni a la primera columna de la imagen se reduce a ocho accesos a memoria y cuatro operaciones matemáticas.

### 4.1.5. VERSIÓN 5

**CÁLCULO DEL MAPA DE DISPARIDAD MEDIANTE LA SUPERPOSICIÓN DE TRES VENTANAS DE AGREGACIÓN**

Al igual que la versión 4, este algoritmo es una evolución de la Versión 3 donde, manteniendo el cálculo simultáneo de ambas disparidades, se busca optimizar el proceso de obtención del coste de agregación.

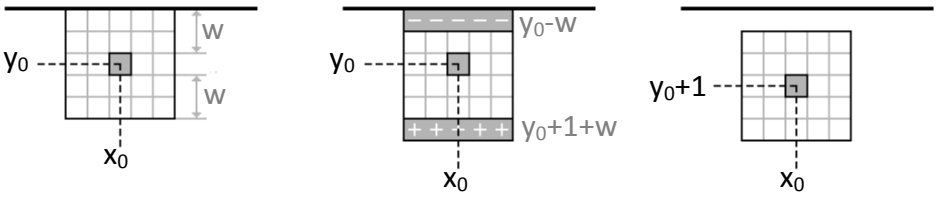



Figura 4.11 Solapamiento entre dos ventanas con la coordenada *x* correlativa.

La primera columna (columna izquierda) se calcula de forma análoga a la expuesta en la Versión 2. Es decir, el coste de agregación de la ventana superior de dicha columna se obtiene sumando las AD de todos sus píxeles. Las demás ventanas de la primera columna se calculan mediante superposición, restando al valor del coste de la ventana previa la contribución de la fila superior, privativa de la anterior ventana, y sumándole el de la inferior, que únicamente pertenece a la nueva (ver esquema de la figura 4.11).

```
desde j=0 hasta j=n
   coste=coste +abs(im_izda[x0+1+w] -im_dcha[x0+1+w] )
               -abs(im_izda[x0-w]   -im_dcha[x0-w]   )
fin desde
```

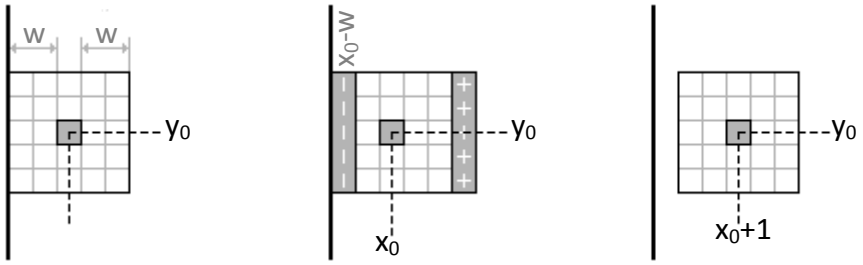



Figura 4.12 Solapamiento lateral entre dos ventanas con la coordenada *y* correlativa.

Para el cálculo del primer elemento de cada una de las columnas restantes se hace uso del solapamiento entre columnas contiguas respecto del eje *x*. De acuerdo al bucle anterior, se recorre la ventana en sentido descendente, restando al valor del coste de

la ventana anterior, centrada en ($x_0$,$y_0$) la contribución de los píxeles de la columna ($x_0$-w), que pertenece únicamente a la ventana previa, y sumándole los de la ($x_0$-1-w), privativa de la nueva.

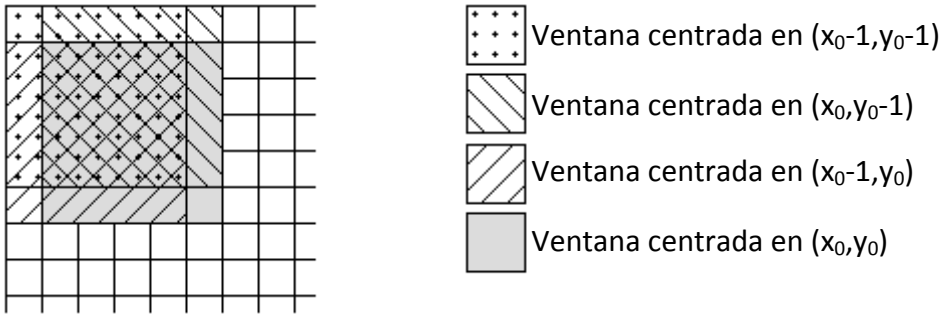



Figura 4.13 Solapamiento entre ventanas.

Para el cálculo de la disparidad del resto de ventanas de la imagen se usa la superposición de acuerdo al esquema 4.13: a partir del coste de agregación calculado para las tres ventanas circundantes centradas en ($x_0$-1,$y_0$-1), ($x_0$,$y_0$-1) y ($x_0$-1,$y_0$) respectivamente, se obtiene el valor de la nueva ventana centrada en ($x_0$,$y_0$):

$$coste[x_0,y_0]=-coste[x_0-1,y_0-1]+coste[x_0,y_0-1]-coste[x_0-1,y_0]+AD[1]-AD[2]-AD[3]+AD[4]$$

Donde las coordenadas de las AD están referidas a las cuatro esquinas de la superficie definida por las cuatro ventanas, esto es:

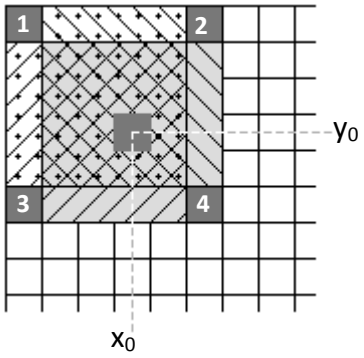



Figura 4.14 Detalle del solapamiento entre ventanas.

Puesto que para el cálculo de una columna hacen falta los costes de agregación de las ventanas superiores de ese columna y los de la anterior, éstos se almacenan en una matriz siguiendo el modelo indicado en la figura 4.15. De esta forma se evita sobrescribir los datos útiles y se minimiza la memoria usada.

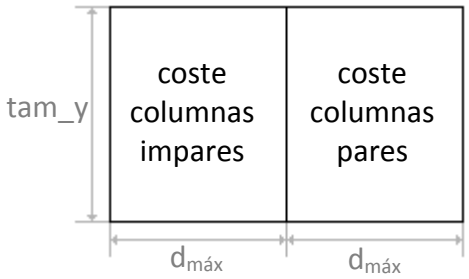



Figura 4.15 Dimensiones y distribución de la matriz de coste.

Para este desarrollo, el coste computacional de calcular el coste de agregación entre un par de ventanas no pertenecientes ni a la primera fila ni a la primera columna de la imagen (el peso del resto de las ventanas se considera despreciable) es, como se deduce del bucle siguiente, de doce accesos a memoria y diez operaciones matemáticas.

```
coste[x0,y0]=-coste[x0-1,y0-1]+coste[x0,y0-1]-coste[x0-1,y0]
             +AD(imagen_izda[1]-imagen_dcha[1])
             -AD(imagen_izda[2]-imagen_dcha[2])
             -AD(imagen_izda[3]-imagen_dcha[3])
             +AD(imagen_izda[4]-imagen_dcha[4])
```

Al igual que en el anterior desarrollo, se observa que a lo largo del algoritmo se calcula cada AD en más de una ocasión. Por evitarlo, se calculan todas las AD al comienzo del algoritmo, almacenándose de forma análoga a la expuesta en el capítulo 4.1.4.

```
coste[x0,y0]=-coste[x0-1,y0-1]+coste[x0,y0-1]-coste[x0-1,y0]
              +AD[1]-AD[2]-AD[3]+AD[4]
```

Por tanto, el cálculo del coste de agregación de una ventana que no pertenezca a un borde de la imagen se reduce a ocho accesos a memoria y seis operaciones matemáticas. Comparando este resultado con el obtenido en la versión 4, donde se requerían también ocho accesos a memoria, pero sólo cuatro operaciones matemáticas, se concluye que la versión óptima es la 4 (este resultado se corrobora en el capítulo de resultados). El presente algoritmo, por contra, requiere menor uso de memoria, siendo el indicado para su implementación en dispositivos con este tipo de limitación.

## 4.2. DETECCIÓN DE OBSTÁCULOS

Una vez calculado el mapa denso de disparidad, donde se almacena la información de la profundidad de cada punto de la imagen, se busca detectar los obstáculos que se encuentran delante del vehículo. Para llevar a cabo esta detección, se va a utilizar la técnica de la *u-v disparity*, introducida anteriormente (en el capítulo 3.5), formada por las etapas que a continuación se enumeran.

### 4.2.1. U_DISPARITY

Como se ha expuesto en el capítulo previo, la imagen *u_disparity* la forman los histogramas de cada columna del mapa de disparidad. En ella, los obstáculos situados perpendicularmente aparecen representados como líneas horizontales, como se observa en la imagen 4.16.

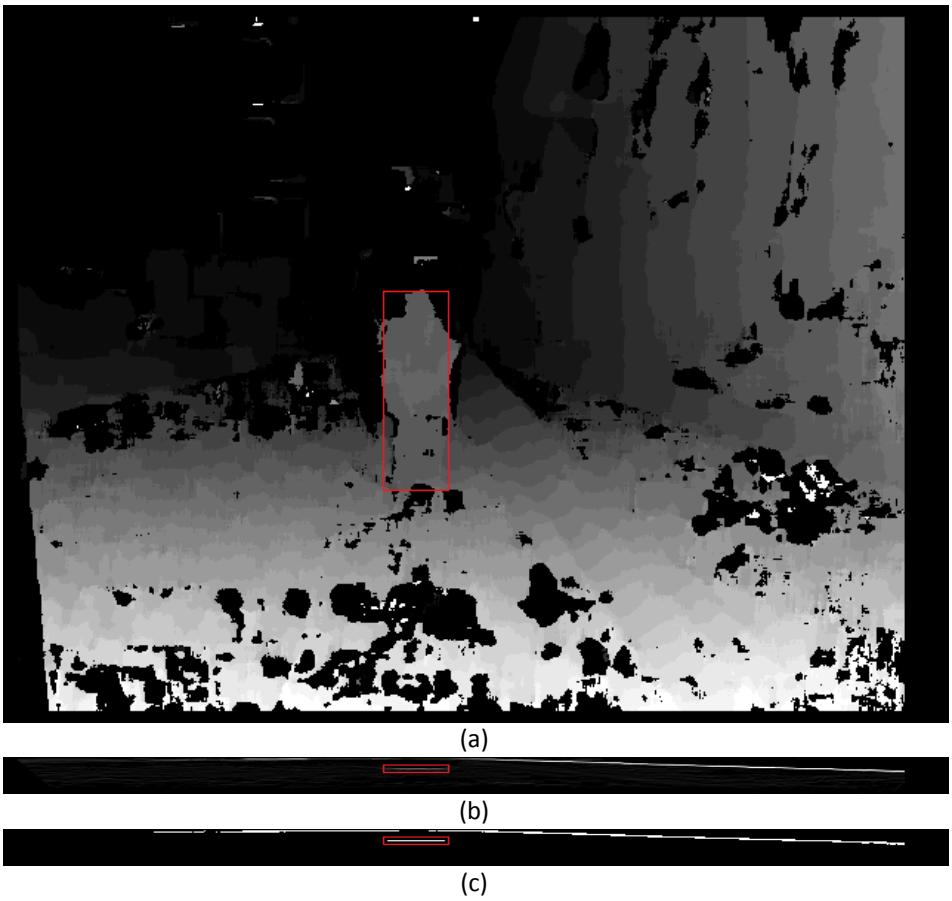



Figura 4.16 (a) Mapa de disparidad (imagen equalizada), (b) *u_disparity*, (c) *u_disparity* umbralizada.

La imagen *u_disparity* se umbraliza para detectar los obstáculos de mayor altura (en píxeles) que el valor umbral. De esta forma se obtiene un esquema de la situación de los obstáculos en términos de u y de la disparidad, sin tener información de la coordenada *v*.

### 4.2.2. MAPA DE OBSTÁCULOS

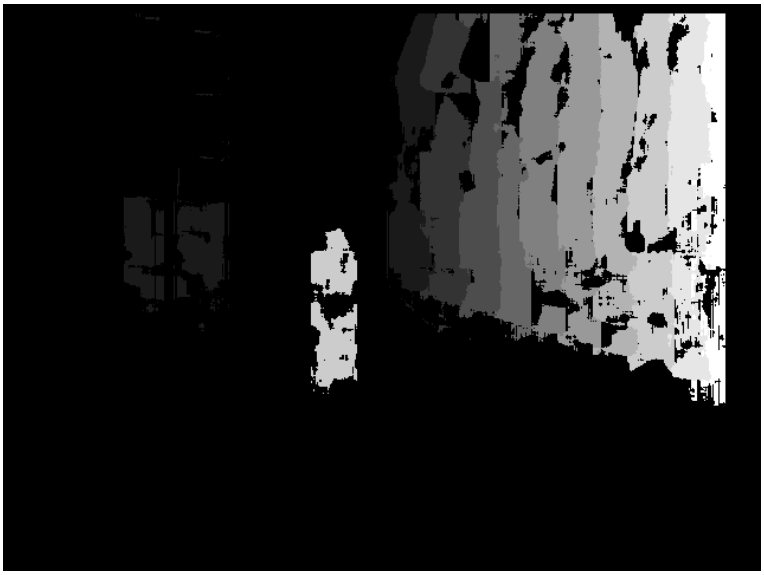
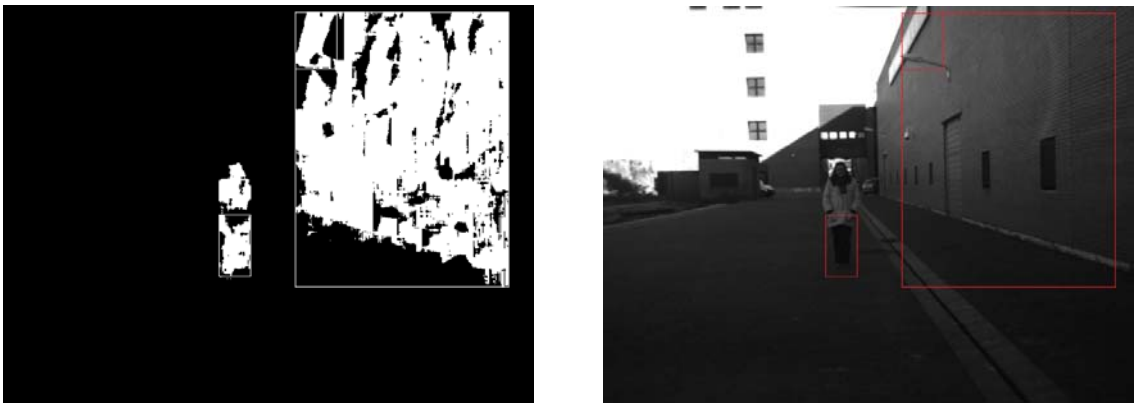


Figura 4.17 Mapa de obstáculos (imagen equalizada).

Para calcular el mapa de obstáculos, mostrado en la figura 4.17, se parte de una *u_disparity* umbralizada, donde sólo aparecen reflejados los obstáculos cuya altura, en píxeles, sea mayor que el valor umbral. Para cada uno de estos píxeles ($u_0$,$d_0$) pertenecientes a un obstáculo de la *u_disparity*, se estudia la columna $u_0$ del mapa de disparidad y, aquellos píxeles cuyo valor de disparidad coincida con $d_0$, son considerados parte del obstáculo, almacenándose como tales en el mapa de obstáculos.

### 4.2.3. DETERMINACIÓN DE LAS REGIONES DE INTERÉS SOBRE LOS OBSTÁCULOS

Como se detalló en el capítulo 3.7, se desea poder determinar regiones de interés sobre cada uno de los obstáculos para una posible etapa de clasificación posterior, como la presentada en [30].



(a) (b)

Figura 4.18 (a) Resultado del *Blob Analysis*: identificación de tres obstáculos en la imagen, (b) regiones de interés determinadas.

El primer paso consiste en umbralizar el mapa de obstáculos anteriormente obtenido. Según el valor del umbral se determina hasta qué distancia se van a buscar los obstáculos. A continuación, se aplica un *Blob Analisys*, figura 4.18 (a). Como resultado se obtiene el número de obstáculos identificados (blobs) y las coordenadas del rectángulo que determina la región de interés que los engloba.

En la figura 4.18 se observa que, para obstáculos de gran tamaño como pueda ser la pared, la localización del área de interés resulta poco precisa. Para paliarlo, se incluye una nueva etapa en el algoritmo.

A partir del mapa de obstáculos, figura 4.19 (a), se obtiene una máscara con los bordes presentes en la imagen (b). En este caso, por simplicidad, la obtención se realiza mediante el algoritmo de Canny, implementado en las librerías OpenCv. La máscara obtenida se aplica sobre la imagen (c), consiguiendo que las zonas con distinto valor de disparidad aparezcan como obstáculos separados. De esta forma, el algoritmo engloba con diversas ROIs cada uno de los tramos de la pared (d).

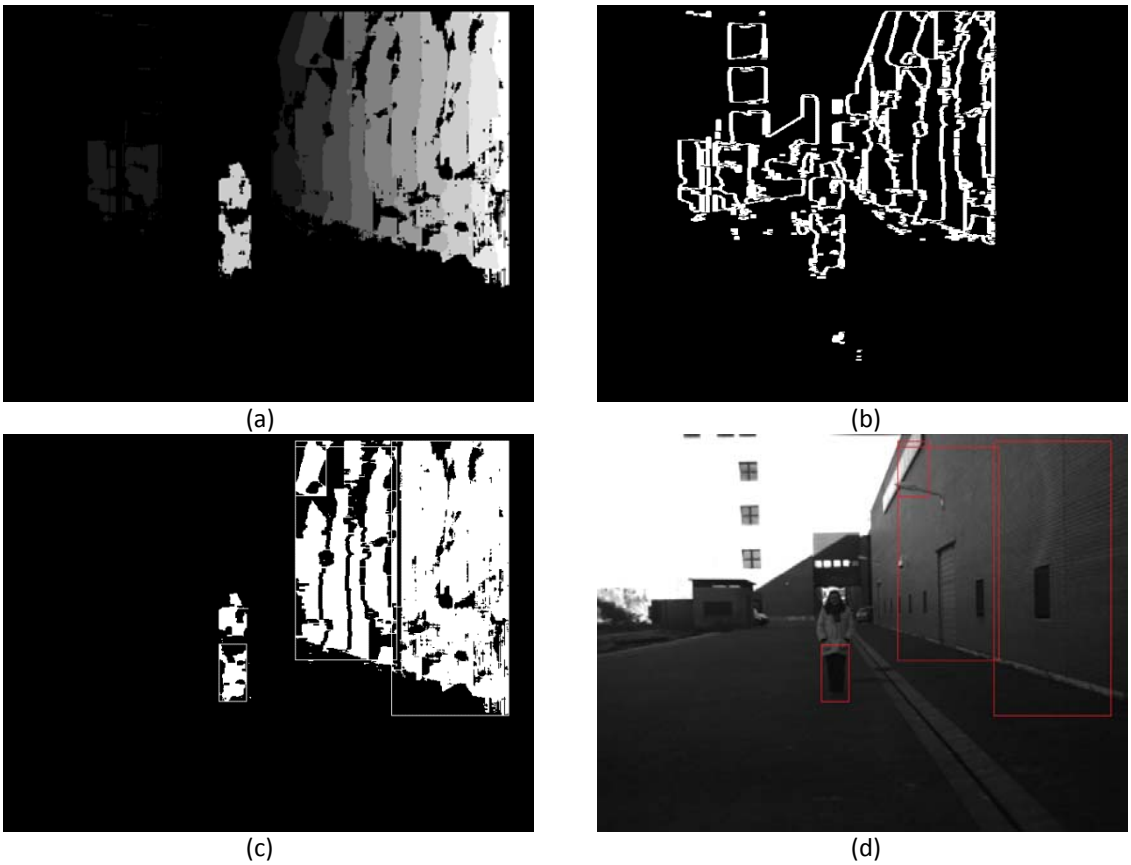



Figura 4.19 (a) Mapa de obstáculos (imagen equalizada), (b) bordes presentes en el mapa de obstáculos, (c) resultado del *Blob Analysis*: identificación de cuatro obstáculos en la imagen, (d) regiones de interés determinadas.

Al finalizar esta etapa del algoritmo, por tanto, se dispone de información sobre el número de obstáculos y su situación en la imagen.

## 4.3. LOCALIZACIÓN DE OBSTÁCULOS

Una vez conocido el número de obstáculos y su ubicación en la imagen, la siguiente etapa es calcular la localización en coordenadas del mundo (W,U,V) de dichos obstáculos. Esta localización se puede llevar a cabo mediante dos métodos. En primer lugar se expone el método basado en los valores de disparidad; a continuación, el basado en el *road profile*.

### 4.3.1. MAPA LIBRE

Como primer paso para la localización de los obstáculos se crea el mapa libre, figura 4.20. Se trata de un nuevo mapa denso de disparidad donde aparecen únicamente aquellas zonas libres de obstáculos, esto es, aquellas que no forman parte de los mismos y que, por consiguiente, normalmente corresponden a la calzada delante del vehículo. Para su cálculo se aprovecha el bucle donde se obtiene el mapa de obstáculos, almacenando todos aquellos píxeles no considerados obstáculos.

(a)

(b)

Figura 4.20 (a) Mapa de disparidad (imagen equalizada), (b) mapa libre (imagen equalizada).

### 4.3.2. OBTENCIÓN DEL *ROAD PROFILE* A PARTIR DE LA *V_DISPARITY*

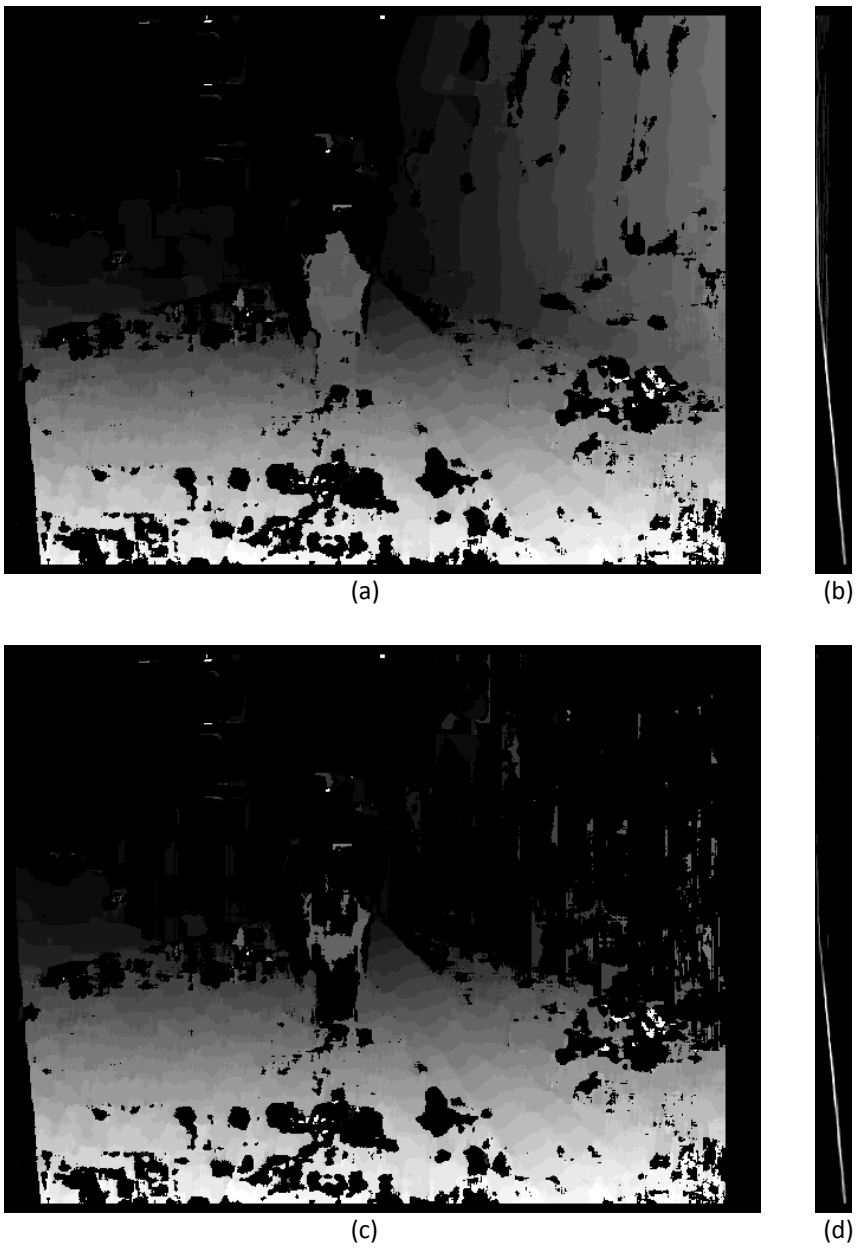
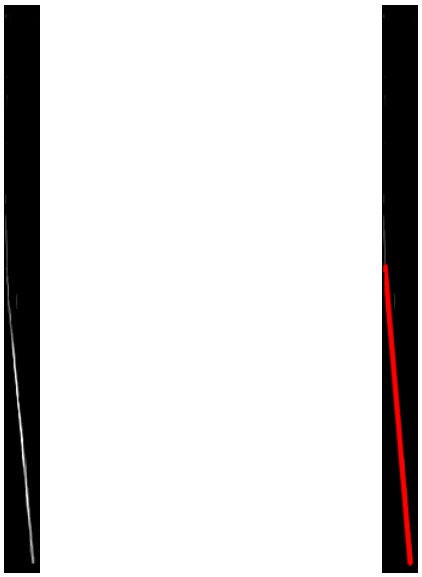


Figura 4.21 (a) Mapa de disparidad; (b) *v_disparity* calculada a partir del mapa de disparidad; (c) mapa libre; (d) *v_disparity* calculada a partir del mapa libre.

En principio, el *road profile* se puede calcular a partir de la *v_disparity* del mapa de disparidad. Sin embargo, usar el mapa libre eleva las probabilidades de que la línea más significativa de la *v_disparity* sea dicho *road profile* (recta oblicua), facilitando su identificación, como se puede comprobar en la imagen superior (figura 4.21 (d)). A continuación, se analizan algunas situaciones en las que el mapa libre resulta imprescindible para la correcta detección del *road profile.*



(a) (b)

Figura 4.22 (a) *v_disparity*, (b) *v_disparity* con la recta resultante de la transformada de Hough.

Al aplicar la transformada de Hough sobre la *v_disparity* se obtiene la recta (definida por las coordenadas de sus extremos) que es la que más puntos de la imagen contiene (ver figura 4.22). Esta línea, definida por la ecuación (22), la más importante, en condiciones normales corresponde al *road profile*, siendo b el horizonte teórico.

$$v = m \cdot d + b \qquad (22)$$

A continuación se analizan algunas situaciones en las que el mapa libre permite la correcta detección del *road profile* [29], contraponiendo los resultados de la transformada de Hough a los obtenidos en caso de calcular la *v_disparity* a partir del mapa de disparidad. El caso más común para que se genera una mala detección del *road profile* en entornos urbanos se produce cuando aparecen uno o más obstáculos de gran tamaño delante del vehículo. En la figura 4.23 se muestran distintos ejemplos de estas situaciones.

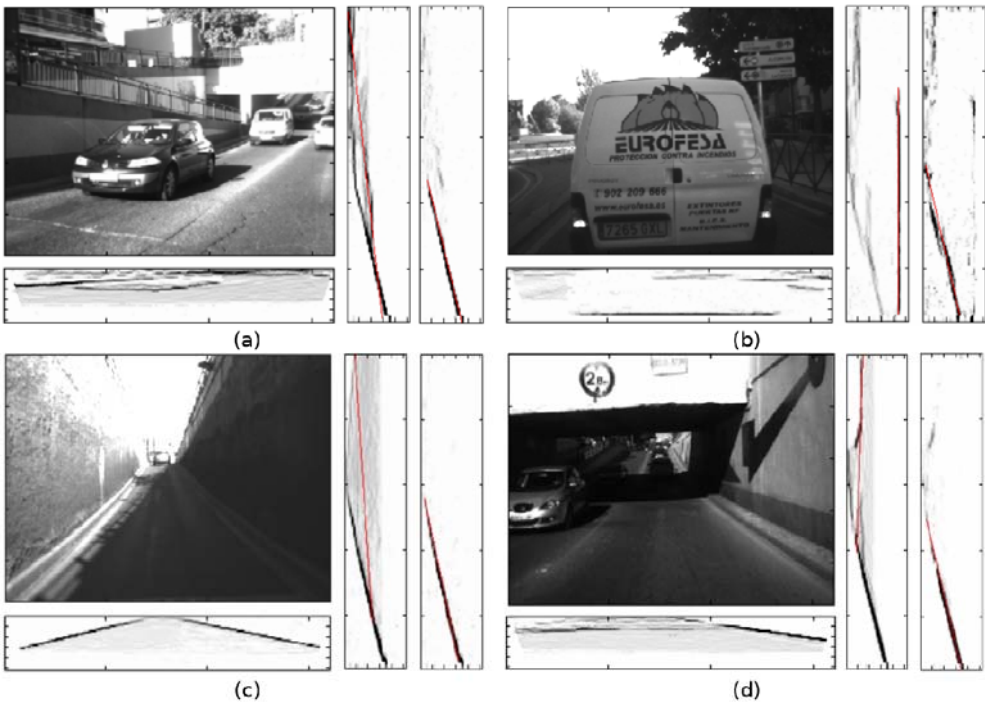



Figura 4.23 Ejemplos de obtención del *road profile* en diferentes casos en entornos urbanos: (a) vehículo y pared, (b) atasco, (c) obstáculo a ambos lados, (d) ejemplo de obstáculo elevado: un túnel. La línea roja marca el *road profile* identificado en cada caso.

Las imágenes (a) y (b) representan dos ejemplos de obstáculos delante del vehículo. Debido al tamaño, en el caso (a), y la proximidad, en el (b), los obstáculos aparecen en la *v_disparity* obtenida a partir del mapa de disparidad (izda) como líneas verticales de gran tamaño, dificultando la identificación del *road profile*. Se observa como, al calcular la *v_disparity* del mapa libre (dcha), únicamente aparece una recta oblicua, que corresponde al *road profile*.

En la imagen (c) se analiza el efecto de la presencia de obstáculos a ambos lados del vehículo sobre la *v_disparity*. De nuevo se comprueba que, al calcularla a partir del mapa de disparidad, las paredes aparecen en la *v_disparity* como una sucesión de rectas verticales de gran tamaño que imposibilitan la identificación del *road profile*. Usando de la transformada de Hough del mapa libre, por contra, la recta que define el perfil del suelo se observa inequívocamente, habiendo sido las paredes eliminadas de la *v_disparity.*

Por último, se comparan los resultados obtenidos para un obstáculo elevado (d). Si bien en este caso las diferencias no son tan obvias, se consigue matizar lo suficiente la

contaminación introducida por el obstáculo en la *v_disparity* como para posibilitar la identificación del *road profile.*

A la vista de los resultados precedentes, se ha decidido implementar el algoritmo para calcular el *road profile* a partir del mapa libre.

### 4.3.3. LOCALIZACIÓN DE OBSTÁCULOS MEDIANTE LA DISPARIDAD

Para hallar la distancia a la que se encuentran los obstáculos del vehículo, se puede usar la ecuación estéreo (23), definida en el capítulo 3.1.3.

$$W = f \cdot B / d \qquad (23)$$

donde P(W,U,V) representa un punto del espacio y p(u,v) su proyección correspondiente en la imagen izquierda, d es la disparidad, f la distancia focal y B la distancia entre las cámaras (denominada *baseline*).

Mediante la ecuación (23) se pueden localizar, es decir, determinar a la distancia que se encuentran, todo tipo de obstáculos, elevados y no elevados, simplemente introduciendo el valor de la disparidad.

### 4.3.4. LOCALIZACIÓN DE OBSTÁCULOS MEDIANTE EL *ROAD PROFILE*

En esta etapa los obstáculos, en primer lugar, se clasifican como elevados y no elevados [28]. Se consideran obstáculos no elevados aquellos que están apoyados en el suelo, como por ejemplo, los peatones. Las señales de tráfico, cuyo soporte, al ser tan fino no se detecta como obstáculo, constituyen un buen ejemplo de impedimento elevado.

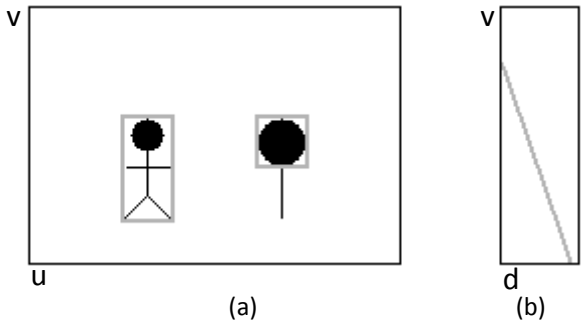



Figura 4.24 (a) Mapa de disparidad, (b) *road profile*.

Puesto que el *road profile* establece la relación entre la posición en la imagen (coordenada *v*) y el valor de disparidad del suelo, se concluye que cualquier obstáculo que también cumpla esta relación está apoyado en el mismo. Se define como disparidad teórica la disparidad que, en función de su posición en la imagen (coordenada v), debería tener un obstáculo que esté apoyado en el suelo. Y se estima cruzando la coordenada inferior del obstáculo con el *road profile* como se indica en la figura 4.25.

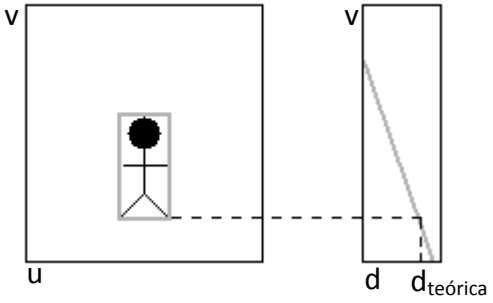



Figura 4.25 Esquema de la estimación de la $d_{teórica}$.

Si la $d_{teórica}$ coincide, con un determinado margen de error, con la d del obstáculo, que se puede estimar como el valor medio de los valores de disparidad dentro de la región de interés, se concluye que se trata de un obstáculo no elevado.

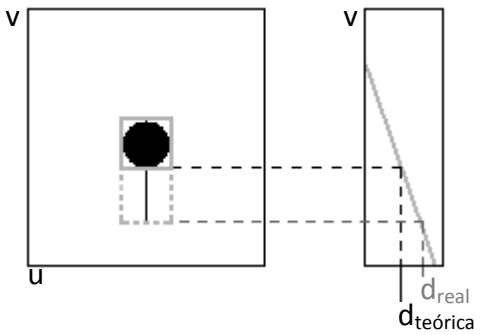



Figura 4.26 Interpolación entre el mapa de disparidad y el *road profile*.

Si $d_{obstáculo} > d_{teórica}$, se trata de un obstáculo elevado. En la imagen 4.26 se justifica esta distinción. Al obstáculo, de acuerdo a su posición en el suelo (b), tiene una disparidad, $d_{real}$. Sin embargo, si se considera el obstáculo como elevado (a), su disparidad no coincide con el valor que toma el suelo para esa posición, $d_{teórica}$.

Se ha definido el *road profile*, calculado mediante la transformada de Hough, como una línea (22). De acuerdo a todo lo anterior, se observa que para obstáculos no elevados, la disparidad teórica y la disparidad real coinciden. Por tanto la ecuación (22) describe también la relación entre la disparidad y la coordenada vertical (v) de la imagen.

Combinando las ecuaciones (22) y (23), se obtiene que la distancia (W) entre el obstáculo y la cámara puede expresarse en función de la posición del obstáculo en la imagen:

$$W = f \cdot B \cdot m / (v - b) \quad (24)$$

$$U = W \cdot (Cu - u) / f \quad (25)$$

Donde Cu es la coordenada u del centro óptico. Mediante la ecuación (24) sólo se pueden localizar obstáculos no elevados.

La distancia W calculada, sin embargo, no es la distancia entre el obstáculo y el vehículo, sino entre éste y la cámara, como se observa en la figura 4.27.

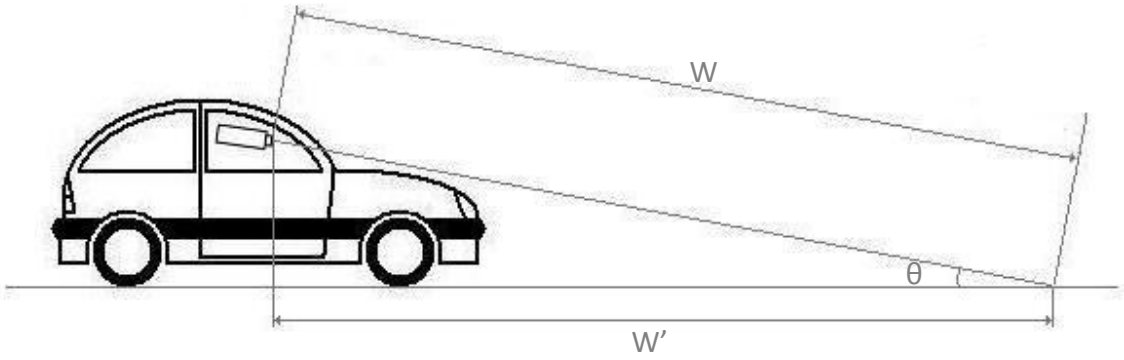



Figura 4.27 Relación entre las distancias cámara-objeto y coche-objeto.

Para obtener la distancia real entre el vehículo y el obstáculo, por tanto, hay que proyectar la distancia anteriormente obtenida sobre el suelo delante del coche:

$$W' = W \cdot \cos\theta \qquad (26)$$

Donde θ, el ángulo que forma la cámara con el suelo, se obtiene mediante un sencillo estudio geométrico [22]:

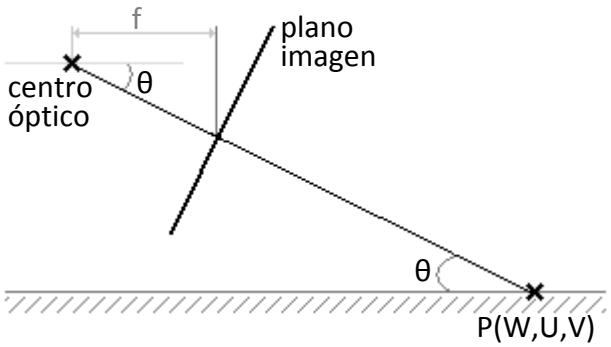



Figura 4.28 Parámetros del modelo *pin-hole* para un punto P(U,V,W) perteneciente al suelo.

De acuerdo a los parámetros definidos en la figura 4.28, el ángulo θ que forma el eje óptico con la horizontal se puede expresar como:

$$\theta = \operatorname{arctag}\left(\frac{b - C_y}{f}\right) \qquad (27)$$

Donde *b* es el horizonte teórico; $C_y$, la coordenada *y* del centro óptico de la imagen y *f* la distancia focal.

### 4.3.5. RESULTADOS EXPERIMENTALES

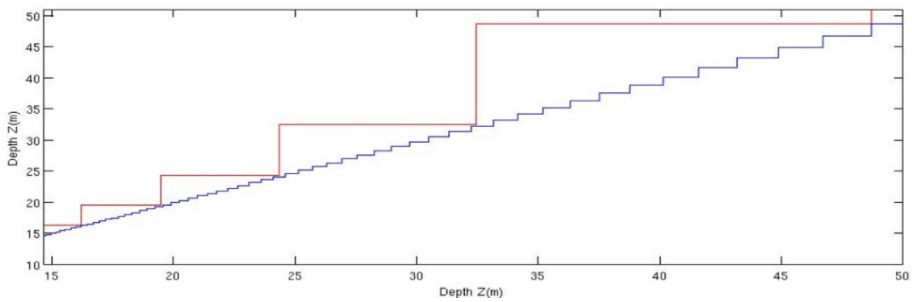



Figura 4.29 Comparación de la resolución de la posición usando los valores de la disparidad (línea superior) y usando el *road profile* (inferior).

Notar que la disparidad puede tomar valores entre 0 y 30. Por contra, la coordenada v del *road profile* tiene una variabilidad típica de 200 píxeles (entre el horizonte b y 480). Esto hace que el segundo método tenga una resolución mayor para localizar los obstáculos como se observa en el gráfico de la figura 4.29. Por tanto, se va a usar la localización mediante el *road profile* siempre que sea posible (es decir, para los obstáculos no elevados).

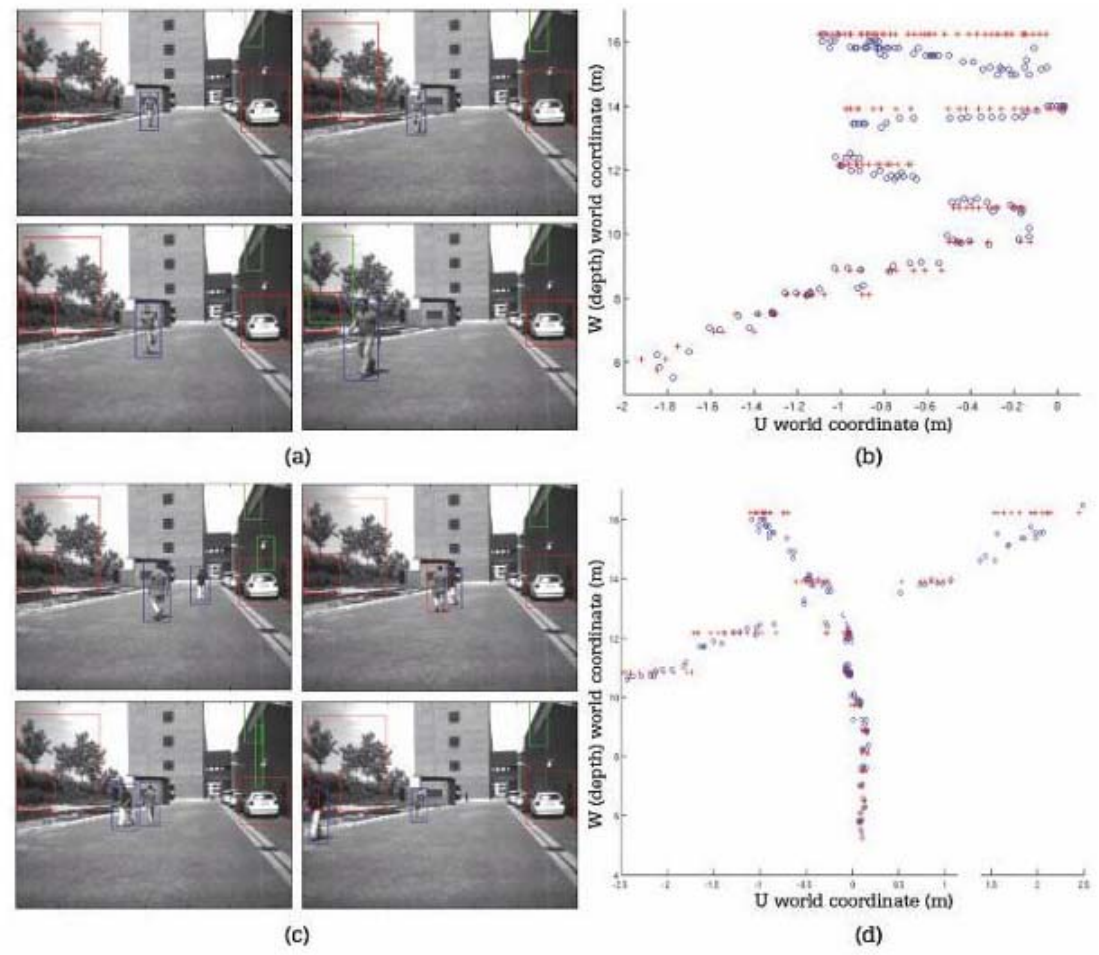



Figura 4.30 Test de localización de obstáculos en un entorno urbano. (a) Diversas imágenes de un peatón realizando un recorrido en zig-zag frente al vehículo; (b) resultados de su localización (en rojo, mediante el método de la disparidad; en azul, a través del *road profile*); (c) recorrido de dos peatones que se cruzan frente al vehículo; (d) resultados de la localización para el cruce de dos vehículos.

Para comprobar la robustez y fiabilidad del algoritmo de localización, se ha llevado a cabo tests en entornos urbanos, un ejemplo de estos experimentos se muestra en la figura 4.30. En primer lugar, un peatón realiza una aproximación al vehículo en zig-zag (a). Los resultados de la localización se muestran en la gráfica adjunta (b), apareciendo en rojo los calculados mediante el método de la disparidad y en azul los obtenidos me-

diante el *road profile*. Se observa como el segundo método es más preciso, permitiendo determinar mejor la trayectoria del peatón. En el método de la disparidad, por contra, debido a su poca resolución no se puede determinar correctamente la trayectoria que describe el peatón.

En el segundo caso, dos peatones se cruzan frente al vehículo (c). Los resultados de la localización (d) demuestran de nuevo la falta de resolución del método de la disparidad frente al del perfil del suelo.

# 5. RESULTADOS

En el presente capítulo se analizan los tiempos de cómputo tanto de las diversas versiones del algoritmo de cálculo del mapa denso de disparidad, expuestas en el capítulo 4.1, como de las etapas de detección y localización de obstáculos.

La comparación de las diversas versiones del algoritmo de cálculo de disparidad tiene como objetivo cuantificar la mejora, en términos del tiempo de cómputo, de las diversas evoluciones. Así mismo, se demuestra que la versión óptima es la cuarta, implementada en el algoritmo de detección y localización de obstáculos.

## 5.1. ANÁLISIS DEL ALGORITMO DE CÁLCULO DE LA DISPARIDAD

| | **VERSIÓN** | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 4 opt. | 5 | 5 opt. |
| **Debug (seg.)** | 42,656 | 4,719 | 4,860 | 0,391 | 0,343 | 0,625 | 0,359 |
| **Release (seg.)** | 35,453 | 3,922 | 4,093 | 0,328 | 0,250 | 0,578 | 0,281 |

Tabla 5.1 Tiempo de cómputo de cada una de las versiones del algoritmo de cálculo del mapa denso de disparidad (en segundos) donde, en las versiones 4 y 5 optimizadas, se calculan todas las AD al principio del algoritmo.

En la tabla 5.1 se recogen los tiempos de cómputo de cada una de las versiones del algoritmo del cálculo de la disparidad para parámetros estándar[2]. Se denominan versiones 4 y 5 aquellas definidas en el capítulo 4.1 donde se comparan los pares de píxeles a medida que se usan. En las versiones 4 y 5 optimizadas, por contra, se calculan todas AD para cada uno de los posibles valores de disparidad al principio, en lugar de calcularlas a medida que se usan, evitándose de este modo repetir el cálculo.

Los test han sido realizados en un equipo con un procesador Intel® CoreTM2 Quad Q9650 a 3 GHz con FSB 1.333 MHz, memoria 4 x 2GB DDRII 800MHz SDRAM y sistema operativo Windows XP.

[2] Se consideran parámetros estándar una imagen de tamaño 640x480px., una ventana de 17x17px. y una $d_{máx}$=30.

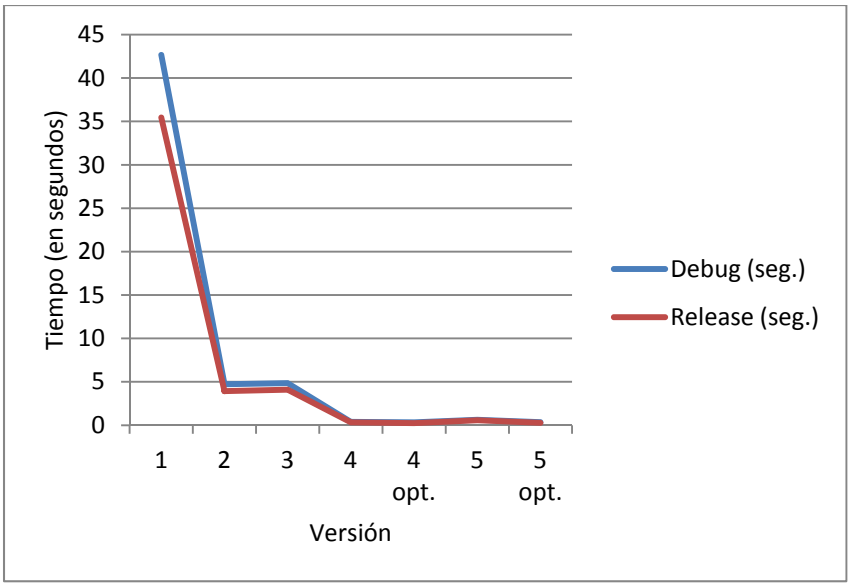



Figura 5.1 Tiempo de cómputo de las diversas versiones del algoritmo de cálculo del mapa denso de disparidad.

En la figura 5.1 se observa como, entre la primera versión del algoritmo de cálculo de la disparidad y la última, se consigue una reducción del tiempo de cómputo de más del 1.000%.

Además, se demuestra como, entre las versiones 2 y 3, apenas se produce un incremento del tiempo computacional. Se concluye, por tanto, que el cálculo de forma simultánea de la disparidad de la imagen izquierda con respecto a la derecha y la de la derecha con respecto a la izquierda tiene un coste computacional despreciable.

### 5.1.1. VERSIONES 4 Y 5

| | **VERSIÓN 4 (optimizada)** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tamaño imagen (px.) | 320x240 | | | | 640x480 | | | | 1280x960 | | | |
| Tamaño ventana (px.) | 11x11 | 17x17 | 23x23 | 17x17 | 11x11 | 17x17 | 23x23 | 17x17 | 11x11 | 17x17 | 23x23 | 17x17 |
| Disparidad | 0-30 | | | 0-60 | 0-30 | | | 0-60 | 0-30 | | | 0-60 |
| **Debug (seg.)** | 0,078 | 0,078 | 0,079 | 0,156 | 0,343 | 0,343 | 0,344 | 0,656 | 1,359 | 1,375 | 1,375 | 2,625 |
| **Release (seg.)** | 0,062 | 0,063 | 0,062 | 0,125 | 0,250 | 0,250 | 0,250 | 0,485 | 1,015 | 1,015 | 1,016 | 1,937 |

Tabla 5.2 Tiempo de cómputo de la versión 4 optimizada del algoritmo de cálculo del mapa denso de disparidad (en segundos) en función de diversos parámetros (tamaño de imagen, tamaño de ventana, $d_{máx}$).

| | VERSIÓN 5 (optimizada) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tamaño imagen (px.) | 320x240 | | | | 640x480 | | | | 1280x960 | | | |
| Tamaño ventana (px.) | 11x11 | 17x17 | 23x23 | 17x17 | 11x11 | 17x17 | 23x23 | 17x17 | 11x11 | 17x17 | 23x23 | 17x17 |
| Disparidad | 0-30 | | | 0-60 | 0-30 | | | 0-60 | 0-30 | | | 0-60 |
| **Debug (seg.)** | 0,094 | 0,094 | 0,094 | 0,171 | 0,360 | 0,359 | 0,360 | 0,687 | 1,437 | 1,438 | 1,437 | 2,781 |
| **Release (seg.)** | 0,062 | 0,062 | 0,062 | 0,125 | 0,281 | 0,281 | 0,281 | 0,531 | 1,125 | 1,125 | 1,125 | 2,141 |

Tabla 5.3 Tiempo de cómputo de la versión 5 optimizada del algoritmo de cálculo del mapa denso de disparidad (en segundos) en función de diversos parámetros (tamaño de imagen, tamaño de ventana, $d_{máx}$).

En las tablas 5.2 y 5.3 se recogen los tiempos de cómputo de las versiones 4 y 5 respectivamente en función de diversos parámetros. Se ha sombreado, de forma orientativa, el resultado correspondiente a los denominados parámetros estándar.

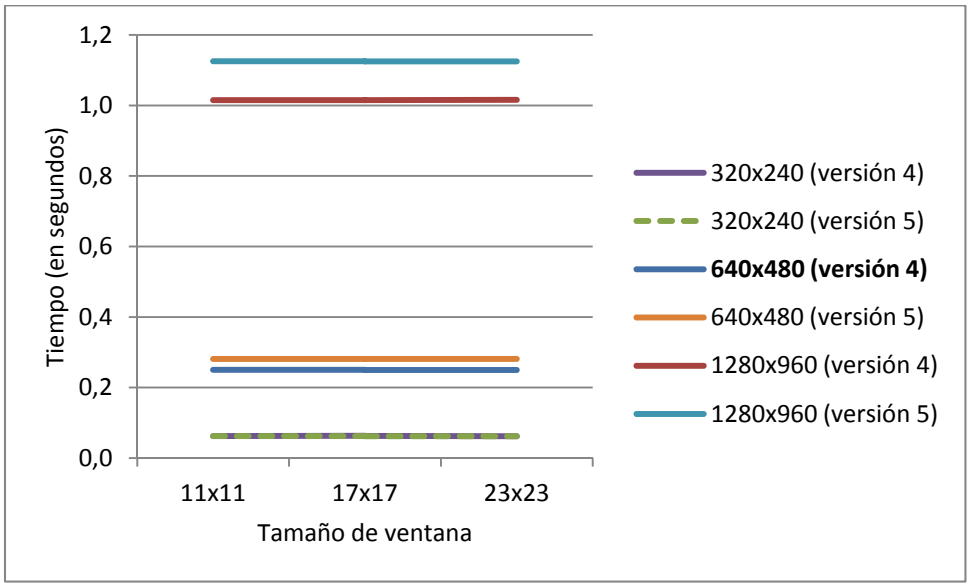
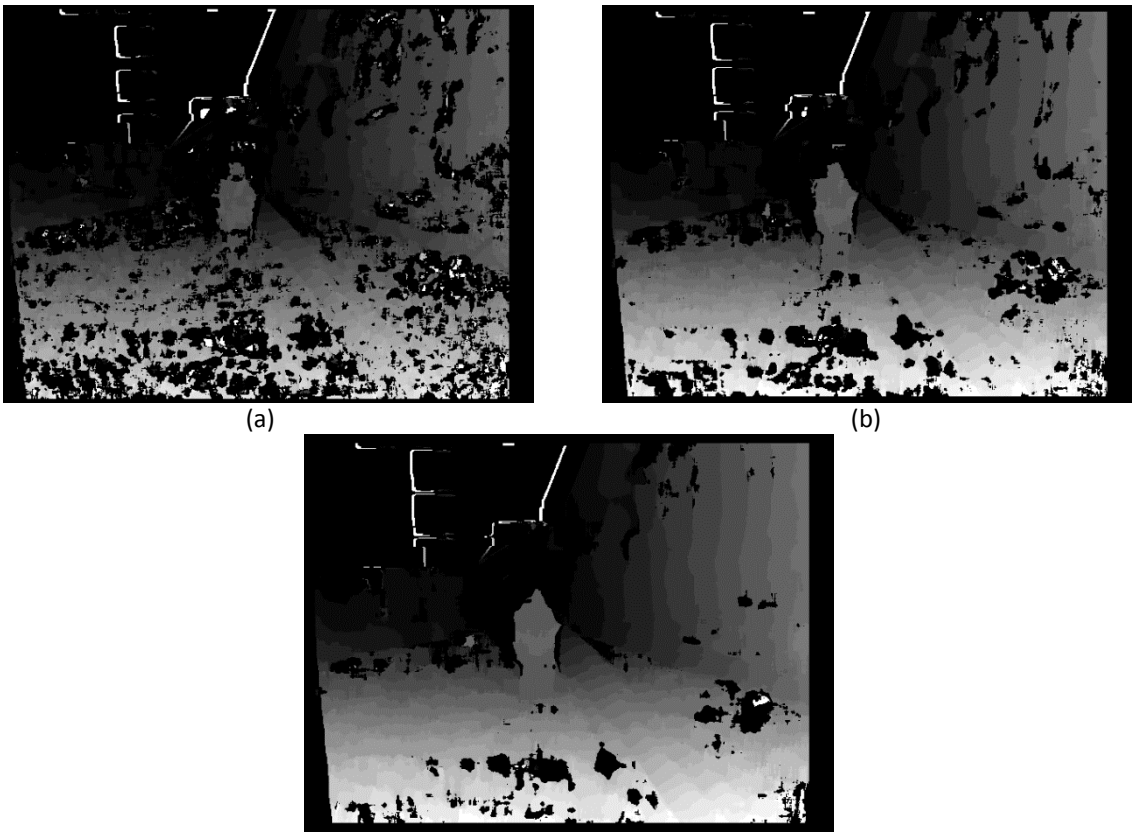



Figura 5.2 Tiempo de cómputo de cada una de las versiones en función del tamaño de ventana para diversos tamaños de imagen.

En la figura 5.2 se demuestra que el tiempo de cálculo para las versiones estudiadas es independiente del tamaño de ventana. Además, cuanto mayor es el tamaño de la ventana, más claramente se observa que el tiempo de cómputo de la versión 4 es menor que el de la 5.



(a) (b)

(c)

Figura 5.3 Mapa denso de disparidad calculado para un tamaño de ventana de (a) 11x11px, (b)17x17px. y (c)23x23px (imágenes equalizadas).

En la figura 5.3 se muestra una comparativa del mapa denso de disparidad calculado para una ventana de tamaño 11x11px. (a), 17x17px. (b) y 23x23px. (c). Se observa como para el tamaño de ventana menor, 11x11px., el mapa denso de disparidad presenta numerosas errores. A pesar de que el aumento del tamaño de la ventana de búsqueda conlleva un descenso de las áreas con errores, debido al incremento en el tiempo de cómputo, se ha elegido implementar en el algoritmo de detección y localización de obstáculos un tamaño medio de ventana (17x17px.).

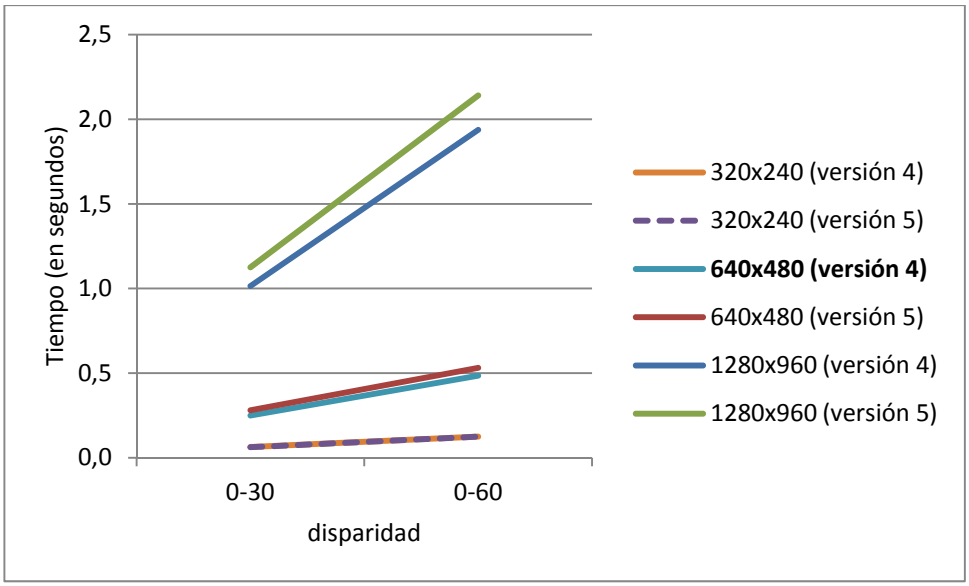



Figura 5.4 Tiempo de cómputo para las versiones 4 y 5 variando la disparidad máxima para diversos tamaños de imagen.

El estudio de la variación de la disparidad máxima se refiere meramente al tiempo de cómputo, ya que, debido a la configuración del sistema estéreo (*baseline*, distancia focal, etc.), la disparidad máxima es de 30 píxeles.

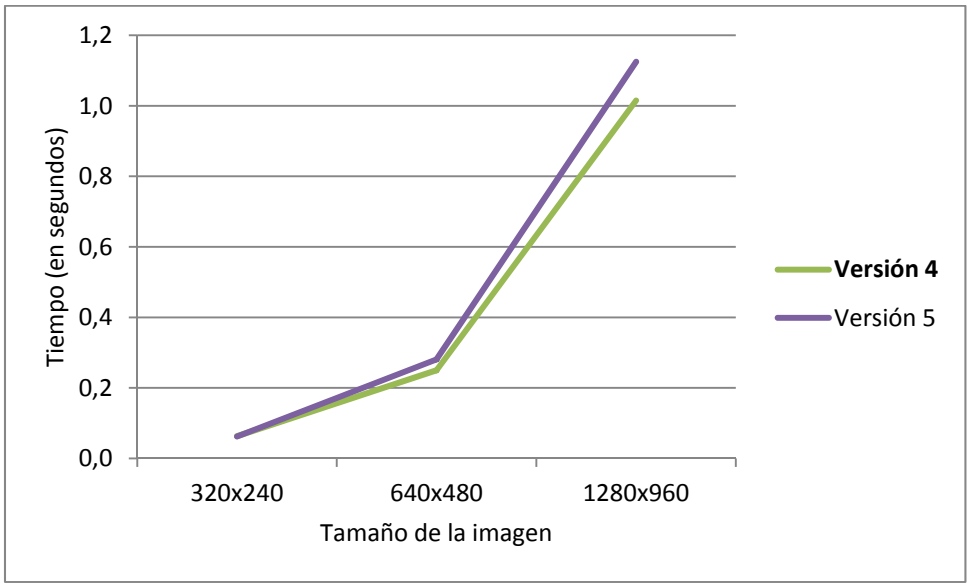



Figura 5.5 Tiempo de cómputo de cada una de las versiones en función del tamaño de la imagen.

En general, de acuerdo a la figura 5.5, el tiempo de cómputo aumenta cuanto mayor es el tamaño de la imagen. En la figura 5.6 se comprueba que dicho aumento se produce de forma lineal con el aumento de la superficie de la imagen.

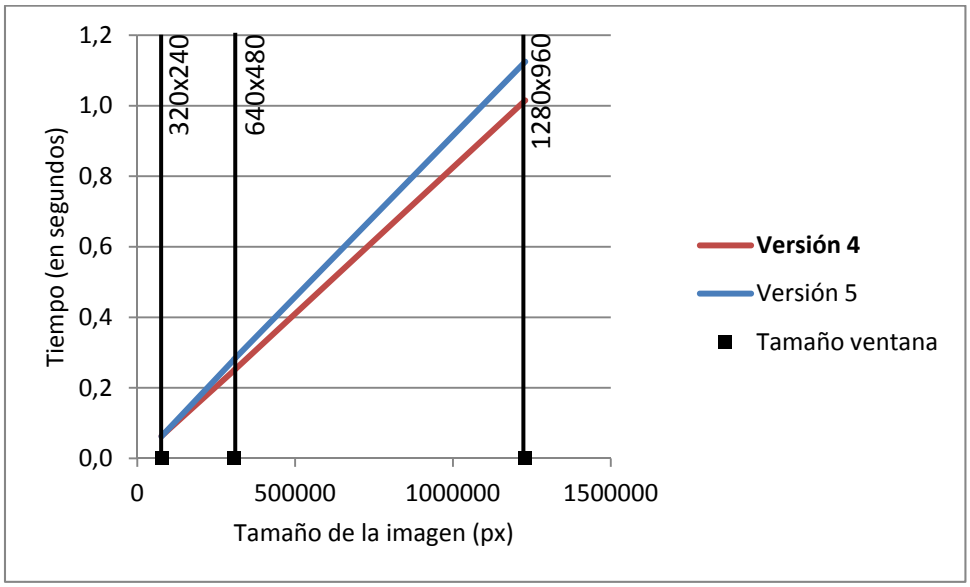



Figura 5.6 Tiempo de cómputo de cada una de las versiones en función del tamaño de la imagen (en px).

## 5.2. ANÁLISIS DEL ALGORITMO DE DETECCIÓN Y LOCALIZACIÓN DE OBSTÁCULOS

A continuación se analiza la influencia sobre el tiempo de cómputo del algoritmo de detección y localización de obstáculos de los parámetros que lo definen.

| Tamaño imagen (px.) | 320x240 | | | 640x480 | | | 1280x960 | | |
|---|---|---|---|---|---|---|---|---|---|
| Tamaño ventana (px.) | 11x11 | 17x17 | 23x23 | 11x11 | 17x17 | 23x23 | 11x11 | 17x17 | 23x23 |
| **Debug (seg.)** | 0,110 | 0,109 | 0,094 | 0,391 | 0,406 | 0,406 | 1,625 | 1,656 | 1,641 |
| **Release (seg.)** | 0,094 | 0,078 | 0,078 | 0,312 | 0,313 | 0,344 | 1,281 | 1,250 | 1,250 |

Tabla 5.4 Tiempo de cómputo del algoritmo de detección y localización de obstáculos (en segundos) en función de diversos parámetros para una disparidad máxima de 30 píxeles.

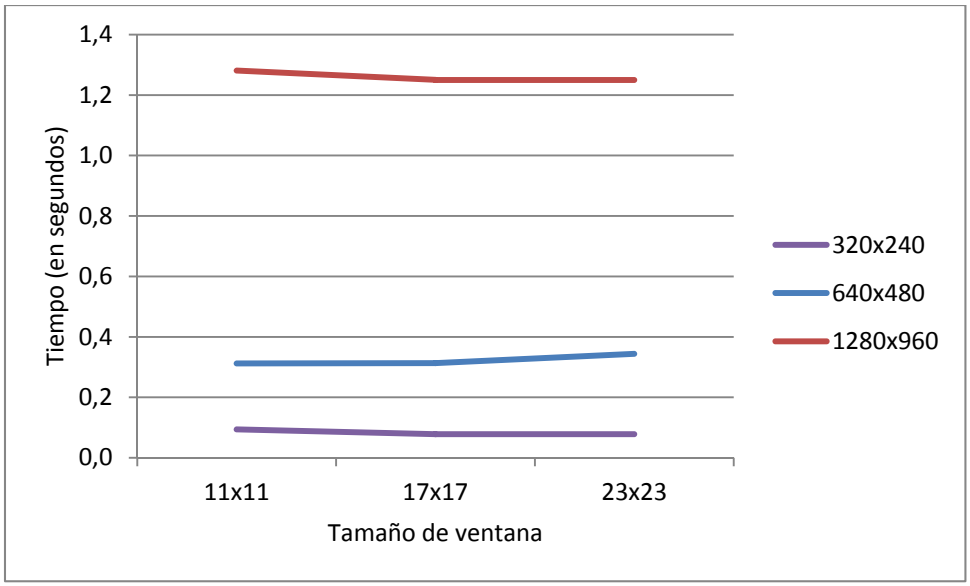



Figura 5.7 Tiempo de cómputo en función del tamaño de ventana para diversos tamaños de imagen.

En la figura 5.7, se puede observar como el algoritmo de detección y localización de obstáculos es independiente del tamaño de ventana. Puesto que en la única etapa que interviene la ventana de búsqueda es en el cálculo del mapa denso de disparidad, este resultado corrobora el obtenido en el apartado 5.1, donde se afirmaba que el algoritmo de cálculo de la disparidad es independiente del tamaño de ventana.

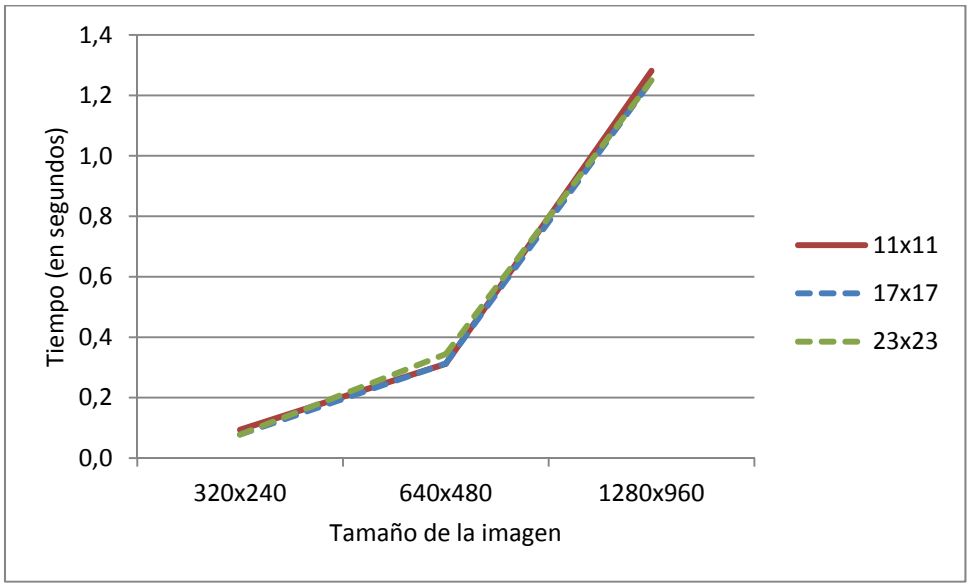



Figura 5.8 Tiempo de cómputo en función del tamaño de la imagen.

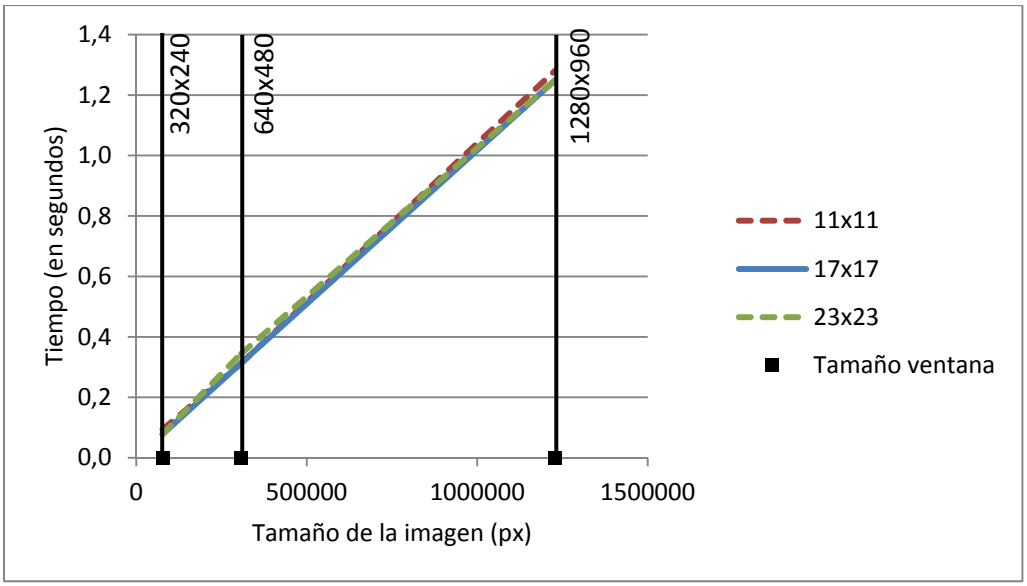



Figura 5.9 Tiempo de cómputo en función del tamaño de la imagen (en px).

Por último, se observa como el tiempo de cómputo aumenta con el tamaño de la imagen (figura 5.8) de forma directamente proporcional (figura 5.9).

En vista de los resultados previos, se observa que el comportamiento del algoritmo de detección y localización de obstáculos al variar los diversos parámetros sigue la pauta descrita por el algoritmo de cálculo del mapa denso de disparidad, ya que, éste constituye la etapa de mayor coste computacional. Por tanto, la elección de los mismos se fija en un tamaño de ventana de agregación de 17x17px. y una disparidad máxima de 30; siendo el tamaño de la imagen captada por la cámara estéreo de 640x480px. Esta configuración nos permite tener un *frame rate* de aproximadamente 3 imégenes por segundo (ver tabla 5.4).

# 6. CONCLUSIONES Y TRABAJOS FUTUROS

## 6.1. CONCLUSIONES

El objetivo principal de este proyecto era la implementación de un algoritmo capaz de detectar y localizar obstáculos en entornos urbanos mediante visión estéreo. Este requisito se ha cumplido con un algoritmo de compromiso entre la robustez de la solución implementada y el tiempo de cómputo requerido. Además, se ha desarrollado una versión alternativa (versión 5) a la versión finalmente implementada (versión 4) para el cálculo del mapa denso de disparidad que requiere una menor necesidad de memoria, posibilitando su implementación en dispositivos con este tipo de limitación (como puede ser una FPGA o GPU (≡ *Graphics Processing Unit*)).

## 6.2. TRABAJOS FUTUROS

El actual algoritmo muestra los obstáculos localizados en un display ubicado en el salpicadero del vehículo, diferenciando los elevados y los no elevados. A continuación se detallan una serie de posibles evoluciones o integraciones que, debido a los requerimientos computacionales, no se han implementado.

Una vez localizados los obstáculos, la evolución lógica es realizar un estudio de las trayectorias de colisión. Para ello, es necesario definir la trayectoria del vehículo (en movimiento) con respecto a un entorno estático. Esto se consigue estimando el Ego-Motion del vehículo [16]. Esta información puede ser obtenida usando un GPS diferencial.

Otra posible ampliación del algoritmo consistiría en la inclusión de una nueva etapa de clasificación de peatones. De acuerdo al estudio [30], es posible, a partir del proyección vertical de los obstáculos no elevados, determinar si se trata o no de un peatón.

# 7. COSTES DEL PROYECTO

El objetivo de este capítulo es presentar una relación aproximada de los costes del proyecto. En primer lugar, se detallan los tiempos dedicados a cada fase del proyecto:

| ETAPA | TIEMPO |
|---|---|
| Estudio y comprensión del problema a resolver | 10 horas |
| Documentación | 30 horas |
| Redacción de la memoria (parte teórica) | 30 horas |
| Implementación del código | 100 horas |
| Pruebas del algoritmo | 20 horas |
| Redacción de la memoria (parte práctica) | 30 horas |
| **TOTAL** | **220 horas** |

Suponiendo un coste aproximado de 35 €/hora (impuestos incluidos) para un programador junior, el precio de la mano de obra sería:

220 horas * 35 €/hora = 7.700€

Además, se han usado los siguientes materiales:

| Concepto | Coste |
|---|---|
| PC estándar | 1.200€ |
| Interfaz salpicadero (Pantalla Xenarc 8 pulgadas) | 400€ |
| Cámara estéreo (Bumblebee de Pointgrey) | 3.000€ |
| Paquete de librerías MIL | 200€ |
| Paquete de librerías OpenCv | gratuito |
| Otro material (cables, soportes, etc.) | 100€ |
| **TOTAL** | **4.900€** |

Suponiendo que en realizar la instalación del equipamiento en el vehículo se tardan 10 horas por parte de un técnico especializado, con un coste de 30 €/hora, el presupuesto se incrementa en:

10 horas * 30 €/hora = 300€

Siendo el coste total del proyecto:

7.700€ + 4.900 € + 300€ = **12.900€**

# 8. BIBLIOGRAFÍA


[1] ABATE, Tom. Robot race suffers quick, ignoble end [en línea]. San Francisco Chronicle [ref. de 14 de marzo de 2004]. Disponible en Web: <http://www.sfgate.com/cgi-bin/article.cgi?f=/c/a/2004/03/14/ROBOT.TMP&ao=all> [Consulta: 20 de septiembre de 2011].

[2] AGUADO ELÍA, Juan. Implementación de un algoritmo de búsqueda de correspondencias estéreo en tiempo real en cuda. Tutores: Rafael Cabeza Laguna y Leonardo De Maeztu Reinares. Universidad Pública de Navarra, Escuela Técnica Superior de Ingenieros Industriales y de Telecomunicación, 2011.

[3] ANUARIO ESTADÍSTICO DE ACCIDENTES 2009. Dirección General de Tráfico. Servicio de Estadística. Observatorio Nacional de Seguridad Vial. N.I.P.O.: 128-09-043-0.

[4] BERTOZZI, Massimo; BROGGI, Alberto; FASCIOLI, Alessandra. Vision-based intelligent vehicles: State of the art and perspectives. ELSEVIER. Robotics and Autonomous Systems 32, 2000, pp.1–16.

[5] BMW. BMW ConnectedDrive. <http://features.bmw.connecteddrive.info/en.html> [Consulta: 8 de agosto de 2011].

[6] BMW. BMW Insights. Technology and innovations. BMW ConnectedDrive. <http://www.bmw.com/com/en/insights/technology/connecteddrive/overview.html> [Consulta: 8 de agosto de 2011].

[7] Bumblebee de Pointgrey, catálogo online. <http://www.ptgrey.com/products/stereo.asp> [Consulta: 10 de octubre de 2011].

[8] CONTINENTAL. Passenger Cars. Chassis & Safety. ContiGuard. <http://www.conti-online.com/generator/www/de/en/continental/automotive/themes/passenger_cars/chassis_safety/contiguard/cs_contiguard_maintopic_en.html> [Consulta: 8 de agosto de 2011].

[9] ELPAIS.COM. Google prueba coches sin conductor [en línea]. El País [ref. de 10 de octubre de 2010]. Disponible en Web: <http://www.elpais.com/articulo/tecnologia/Google/prueba/coches/conductor/elpeputec/20101010elpeputec_1/Tes> [Consulta: 8 de agosto de 2011].

[10] ESCALERA HUESO, Arturo de la. Visión por computador. Fundamentos y métodos. Madrid: Pearson Educación, 2001. 304 p. ISBN: 84-205-3098-0.

[11] Euro NCAP. Our rewards. Ford Active City Stop. <http://www.euroncap.com/rewards/Ford_ActiveCityStop.aspx> [Consulta: 8 de agosto de 2011].

[12] Euro NCAP. Our rewards. Ford Lane Keeping Aid. <http://www.euroncap.com/rewards/ford_LaneKeepingAid.aspx> [Consulta: 8 de agosto de 2011].

[13] Faugeras, O.; Vieville, T.; Theron, E.; Vuillemin, J.; Hotz, B.; Zhang, Z.; Moll, L.; Bertin, P.; Mathieu, H.; Fua, P. Real-time correlation-based stereo: algorithm, implementations and applications. Institut National de Recherche en Informatique et en Automatique, n° 2013, 2003

[14] GONZÁLEZ JIMÉNEZ, Javier. Visión por computador. Madrid: Paraninfo, 2000. 429 p. ISBN: 84-283-2603-4.

[15] GREENE, Kate. Champion Robot Car Declared [en línea]. Technology Review (Published by MIT) [ref. de 5 de noviembre de 2007]. Disponible en Web: <http://www.technologyreview.com/computing/19661/> [Consulta: 20 de septiembre de 2011].

[16] HARRIS, CG. Determination of ego-motion from matched points. 3rd Alvey Vision Conference, 1987. Pp. 189-192.

[17] HAVEit. Final Event. HAVEit Joint System. <http://haveit.lighthouse.gr/LH2Uploads/ItemsContent/122/2_HAVEit_Final_Event_JointSystem_Flemisch.pdf> [Consulta: 8 de agosto de 2011].

[18] HAVEit. Highly Automated Vehicles for Intelligent Transport. <http://haveit.lighthouse.gr/LH2Uploads/ItemsContent/73/Haveit-Brochure.pdf> [Consulta: 8 de agosto de 2011].

[19] HAVEit. Media. Driving without a Driver – Volkswagen presents the “Temporary Auto Pilot”. <http://www.haveit-eu.org/displayITM1.asp?ITMID=117&LANG=EN> [Consulta: 8 de agosto de 2011].

[20] HIRSCHMÜLLER, H.; SCHARSTEIN. Evaluation of cost functions for stereo matching. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), Minneapolis, EEUU, Junio 2007.

[21] Image Analysis and Processing. Developer Zone. National Instruments. <http://zone.ni.com/devzone/cda/tut/p/id/3470#toc2> [Consulta: 25 de agosto de 2011].

[22] LABAYRADE, Raphael; AUBERT, Didier; TAREL, Jean-Philippe. Real Time Obstacle Detection in Stereovision on Non Flat Road Geometry Through “V-disparity” Representation. Intelligent Vehicle Symposium, 2002. IEEE. Volume 2, pp. 646-651.

[23] LONG, Zhiling; YOUNANA, Nicolas H. Denoising of images with multiplicative noise curruption. 13th European Signal Processing Conference EUSIPCO 2005, September 4-8, 2005, Antalya, Turkey.

[24] MARK, Wannes van der; GAVRILA, Dariu M. Real-Time Dense Stereo for Intelligent Vehicles. IEEE Transactions On Intelligent Transportation Systems, Vol. 7, No. 1, pp. 38-50, March 2006.

[25] MARKOFF, John. Guided by Computers and Sensors, a Smooth Ride at 60 Miles Per Hour [en línea]. The New York Times [ref. de 9 de octubre de 2010]. Disponible en Web: <http://www.nytimes.com/2010/10/10/science/10googleside.html?scp=1&sq=google%20toyota&st=Search> [Consulta: 8 de agosto de 2011].

[26] MARTÍN CLEMENTE, Alejandro. Generación de mapas de disparidad utilizando cuda. Tutor: Dr. Arturo de la Escalera Hueso, Director: Basam Musleh Lancis. Universidad Carlos III de Madrid, Escuela Politécnica Superior, Departamento de Ingeniería de Sistemas y Automática, 2009.

[27] MARTÍN JIMÉNEZ, José Antonio; ARIAS PÉREZ, Benjamín; GONZÁLEZ AGUILERA, Diego; GÓMEZ LAHOZ, Javier. Procesamiento avanzado de imágenes digitales, 2010-11. Universidad de Salamanca (España). Open Course Ware. 2010.

[28] MUSLEH, Basam; ESCALERA, Arturo de la; ARMINGOL, José María. Real-Time Pedestrian Recognition in Urban Enviroments. Advanced Microsystems for Automotive Applications. Springer, 2011. pp. 139-147.

[29] MUSLEH, Basam; ESCALERA, Arturo de la; ARMINGOL, José María. U-V Disparity Analysis in Urban Environments. EUROCAST 2011. Thirteen International Conference on COMPUTER AIDED SYSTEMS THEORY. IUCTC Universidad de Las Palmas de Gran Canaria. ISBN: 978-84-693-9560-8. Pp. 169-170.

[30] MUSLEH, Basam; GARCÍA, Fernando; OTAMENDI, Javier; ARMINGOL, José María; ESCALERA, Arturo de la. Identifying and Tracking Pedestrians Based on Sensor Fusion and Motion Stability Predictions. MDPI. Sensors 2010, *10*(9); doi:10.3390/s100908028. ISSN 1424-8220. Pp. 8028-8053.

[31] PAJARES MARTINSANZ, Gonzalo; CRUZ GARCÍA, Jesús M. de la. Visión por computador: imágenes digitales y aplicaciones. Madrid: Ra-Ma, 2001. 764 p. ISBN: 84-789-7472-5.

[32] RUSSELL, Steve. DARPA Grand Challenge Winner: Stanley the Robot! [en línea]. Popular Mechanics [ref. de 9 de enero de 2006]. Disponible en Web: <http://www.popularmechanics.com/technology/engineering/robots/2169012> [Consulta: 20 de septiembre de 2011].

[33] SCHARSTEIN, Daniel; SZELISKI, Richard. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. International Journal of Computer Vision, 47(1/2/3):7-42, April-June 2002. Microsoft Research Technical Report MSR-TR-2001-81, November 2001.

[34] VOLVO. S60. Producto y Accesorios. Características y equipamiento. <http://www.volvocars.com/es/all-cars/volvo-s60/details/pages/features.aspx#> [Consulta: 8 de agosto de 2011].

# ANEXO 1 · CÓDIGO DEL ALGORITMO

## 1. TRATAMIENTO DE IMÁGENES CON LAS LIBRERÍAS

A continuación, se detallan los diversos formatos que adquieren las imágenes en los paquetes de librerías usadas (MIL y OpenCV). En general, se asume que la información de la imagen se carga en una matriz de caracteres sobre la cual se trabaja. El proceso es bidireccional, permitiendo mostrar los resultados como una nueva imagen.

### 1.1. LIBRERÍAS MIL

En el algoritmo, las imágenes tomadas por la cámara estéreo se cargan al programa con formato matriz mediante las librerías especializadas MIL. El motivo de la elección es la mayor simplicidad del almacenamiento de las imágenes en las librerías MIL frente a las librerías OpenCv. De esta forma, la imagen se carga como una matriz de tipo unsigned char del mismo tamaño que la imagen original, coincidiendo las coordenadas de los píxeles en la imagen y la matriz.

Una vez establecidos los identificadores y el buffer e inicializado éste último (para más información, consultar el Anexo 1 · Código del Algoritmo), la imagen se carga a formato matriz en dos pasos. En primer lugar se identifica el fichero:

```
MbufLoad("Imagen.tif", MilImage);
```

Donde Imagen.tif es la imagen y MilImage el identificador. A continuación se carga la imagen:

```
MbufGet(MilImage, matriz);
```

Donde se define matriz como:

```
unsigned char* matriz =(unsigned char*)calloc((tam_x*tam_y),sizeof(unsigned char));
```

## 1.2. LIBRERÍAS OpenCV

La transformada de Hough se encuentra implementada en las librerías OpenCv. Dada su complejidad matemática, se ha optado por usar estas librerías de gratuitas. Por tanto, es necesario realizar una conversión entre el formato MIL (que se usa en el resto del algoritmo) y el OpenCv. Existen dos diferencias fundamentales a nivel operativo entre ambos formatos:

- Los datos de las imágenes en formato MIL se almacenan en una matriz de caracteres del mismo tamaño que la imagen. La correspondencia entre un píxel de la imagen y una celda de la matriz es obvia:

  $$\text{imagen}(y_0, x_0) \equiv \text{matrizMIL}(y_0 * \text{tam\_x} + x_0)$$

  En las librerías OpenCv sin embargo, la relación sería:

  $\text{imagen}(y_0, x_0) \equiv \text{matrizOpenCv}(y_0 * \texttt{imagen->widthStep} + x_0 * \texttt{imagen->nChannels})$

  Donde :

  - `widthStep` es el tamaño asignado a la columna de la imagen en bytes (en el presente algoritmo, `widthStep=32`. La columna original tiene 30 px, a lo que hay que sumar uno más por cada canal de color que no se esté usando. Notar que la imagen, por defecto, tiene tres canales de color de los cuales sólo se está usando uno).
  - `nChannels` es el número de canales de color (en este caso, por tanto, `nChannels=1`).
  -

- En OpenCv es necesario definir una cabecera que contenga toda la información de la imagen:

  ```
  IplImage *cvCreateImageHeader(cvSize, int depth, int channels);
  ```

  Donde :

  - `cvSize` es el tamaño de la imagen destino.
  - `depth` corresponde al tamaño de cada píxel, en bits (en este caso, de forma análoga a la imagen MIL, se define cada píxel como unsigned char, esto es, 8 bits).

- `channels` es el número de canales de color (como se indicaba anteriormente, sólo se va a hacer uso de un canal).

## 2. CÓDIGO DEL ALGORITMO

```
using namespace cv;

#include <stdafx.h>
#include <iostream>
#include <cstdlib>

using namespace std;

#include <cv.h>
#include <highgui.h>
#include <cvaux.h>
#include <cxcore.h>

#include <stdio.h>
#include <math.h>
#include <time.h>

#include <mil.h>
#include <conio.h>

//////////////////////////////////////////////////////////////////////

#define tam_x         640     // Ancho de las imagenes estéreo.
#define tam_y         480     // Alto de las imagenes estéreo.
#define tam_coste     30      // Disparidad máxima (=tamaño vector coste).
#define rad_vent      8       // Radio de la ventana de búsqueda (lado
                                 de la ventana de búsqueda=2*rad_vent+1)
#define umbral_u      48      // Valor umbral para la binarización de la
                                 u_disparity.
#define umbral_obs    3       // Valor umbral para la binarización del
                                 mapa obstáculo.
#define umbral_v      30      // Valor umbral para la umbralización de
                                 la v_disparity.
#define umbral_blob   1500L   // Área mínima de cada obstáculo.

//////////////////////////////////////////////////////////////////////

void laplaciana(unsigned char*,int*,unsigned char*);
void disparidad(unsigned char*,unsigned char*,unsigned char*,unsigned
                char*);
void u_disparity(unsigned char*,unsigned char*);
void mapas(unsigned char*,unsigned char*,unsigned char*,unsigned char*);
void v_disparity(unsigned char*,unsigned char*);

//////////////////////////////////////////////////////////////////////

void main(void)
{
  //Establecimiento de los identificadores de las operaciones MIL
```

```
MIL_ID  MilApplication, // Identificador de la aplicación.
        MilSystem,      // Identificador del sistema MIL.
        MilDisplay,     // Identificador del entorno de visualización.
        MilImageRight,  // Identificador de la imagen (1) [tam_x*tam_y].
        MilImageLeft,   // Identificador de la imagen (2) [tam_x*tam_y].
        MilImageDisp,   // Identificador de display.
        MilImageAux1,   // Identificador de la imagen [tam_x*tam_coste].
        MilImageAux2,   // Identificador de la imagen [tam_coste*tam_y].
        BlobResult,     // Identificador del resultado de blobs.
        FeatureList;    // Identificador de las características de los
                            blobs.

// Establecimiento del buffer de la aplicación.
MappAllocDefault(M_SETUP,&MilApplication,&MilSystem,&MilDisplay,M_NULL,
                 M_NULL);

// Inicialización del buffer para la imagen.
MbufAlloc2d(M_DEFAULT,tam_x,tam_y,8L+M_UNSIGNED,M_IMAGE+M_DISP+M_PROC,
            &MilImageLeft);
MbufAlloc2d(M_DEFAULT,tam_x,tam_y,8L+M_UNSIGNED,M_IMAGE+M_DISP+M_PROC,
            &MilImageRight);
MbufAlloc2d(M_DEFAULT,tam_x,tam_y,8L+M_UNSIGNED,M_IMAGE+M_DISP+M_PROC,
            &MilImageDisp);
MbufAlloc2d(M_DEFAULT,tam_x,tam_coste,8L+M_UNSIGNED,M_IMAGE+M_DISP+M_P
            ROC,&MilImageAux1);
MbufAlloc2d(M_DEFAULT,tam_coste,tam_y,8L+M_UNSIGNED,M_IMAGE+M_DISP+M_P
            ROC,&MilImageAux2);

clock_t init, final;
init=clock();

// Inicialización de las matrices de datos de imágenes.
unsigned char* im_izda=   (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* im_dcha=   (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* lp_izda=   (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* lp_dcha=   (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* disp_izda= (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* disp_dcha= (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* u_disp=    (unsigned char*)calloc((tam_x*tam_coste),
                          sizeof(unsigned char));
unsigned char* mapa_obs = (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* mapa_lib = (unsigned char*)calloc((tam_x*tam_y),
                          sizeof(unsigned char));
unsigned char* v_disp =   (unsigned char*)calloc((tam_coste*tam_y)
                          ,sizeof(unsigned char));

// Carga de los ficheros.
MbufLoad("dcha.tif",MilImageRight);
MbufLoad("izda.tif",MilImageLeft);

MbufGet(MilImageRight, im_dcha);
MbufGet(MilImageLeft , im_izda);
```

```
// Se define la máscara usada para el cálculo de la Laplaciana de la
     Gaussiana.

   int lg [25] = {   0, -1, -4, -1,  0,
                    -1, -4,  2, -4, -1,
                    -4,  2, 32,  2, -4,
                    -1, -4,  2, -4, -1,
                     0, -1, -4, -1,  0};

// LAPLACIANA DE LA GAUSSIANA //

laplaciana(im_izda, lg, lp_izda);
laplaciana(im_dcha, lg, lp_dcha);

free(im_izda);
free(im_dcha);

// DISPARIDAD //

disparidad(lp_izda, lp_dcha, disp_izda, disp_dcha);

free(lp_izda);
free(lp_dcha);

// CROSS-CHECKING //

for (int k=0; k<(tam_x*tam_y); k++)
{
   if (abs(disp_dcha[k]-disp_izda[k])>2)
     {disp_izda[k]=0;}
}

free(disp_dcha);

// U_DISPARITY //

u_disparity(disp_izda, u_disp);

// MAPA OBSTÁCULO Y MAPA LIBRE //

mapas(disp_izda, u_disp, mapa_obs, mapa_lib);

free(u_disp);

// V_DISPARITY //

v_disparity(mapa_lib, v_disp);

// TRANSFORMADA DE HOUGH //

// Se realiza la conversión entre el formato de imagen de las MIL
// (formato de origen) y el de las OpenCV (formato en el cual se va
// realizar la transformada de Hough).

// Se define la matriz donde se van a volcar los datos en formato
// OpenCV. Para más información sobre el tamaño de la matriz,
// consultar la memoria, Capítulo 4.3. Tratamiento de las imágenes con
// librerías OpenCv.

char* aux_v =  (char*)calloc(((tam_coste+2)*tam_y),sizeof(char));
```

```
// Se define la cabecera de la nueva imagen.

IplImage *v = cvCreateImageHeader(cvSize(tam_coste,tam_y),8,1);

// IplImage *cvCreateImageHeader(cvSize, int depth, int channels);
//   cvSize   -> Tamaño de la imagen destino.
//   depth    -> Tamaño de cada píxel, en bits.
//   channels -> Número de canales de color.

// En los tres parámetros se han conservado los valores de la imagen
//    origen (v_disp):
//       cvSize = tam_coste x tam_y.
//       depth = unsigned char = 8 bits.
//       channels = 1.

// Se realiza la equivalencia entre los píxeles de la imagen,
// almacenados en formato MIL (v_disp), a formato OpenCv (aux_v).

// for(y) y for(x) recorren todos los píxeles de la matriz origen
// (v_disp), cargando los datos en el lugar indicado de la matriz
// destino (aux_v).

for(int y=0; y<tam_y; y++)
{
  for(int x=0; x<tam_coste; x++)
  {
    aux_v[v->widthStep*y+x*v->nChannels]=v_disp[y*tam_coste+x];

    // Para realizar la conversión de los datos desde formato MIL a
    // formato OpenCV, se requieren dos parámetros de la imagen
    // OpenCV:
    //     widthStep -> Tamaño asignado a la columna de la imagen en
    //       bytes (widthStep=32).
    //   nChannels -> Número de canales de color (nChannels=1).
    // Por tanto, la diferencia entre la matriz de datos en formato
    // MIL y en formato OpenCv, en este caso, serán en dos columnas
    // vacías a la derecha de la matriz.
  }
}

// Se cargan los datos de la nueva matriz calculada a la imagen OpenCv.

v->imageData = aux_v;

// Se define una estructura para poder almacenar datos genéricos de
// forma dinámica en formato OpenCv.

CvMemStorage* datos = cvCreateMemStorage(0);

// Se define una estructura para poder almacenar líneas de forma
// dinámica en formato OpenCv.

CvSeq* lineas = 0;

// Se calcula la transformada de Hough de la imagen v, almacenando los
// resultados en lineas.

lineas = cvHoughLines2(v,datos,CV_HOUGH_PROBABILISTIC,1,CV_PI/180,
                       50,75,20);
```

```
// CvSeq* cvHoughLines2(CvArr* image,void* storage,int method,double rho,
//          double theta,int threshold,double param1=0,double param2=0);
//    image   -> Imagen de origen.
//    storage -> Almacenamiento de las líneas detectadas.
//    method  -> Método de cálculo de la transformada de Hough:
//                  CV_HOUGH_STANDARD      -> Método clásico.
//                  CV_HOUGH_PROBABILISTIC -> Método probabilístico.
//                  CV_HOUGH_MULTI_SCALE   -> Variable multiescala del
//                                            método clásico.
//    rho     -> Distancia de resolución (en píxeles).
//    theta   -> Ángulo de resolución (en radianes).
//    threshold -> Parámetro umbral.
//    param1  -> Para la transformada de Hough probabilística,
//                  longitud mínima de la línea.
//    param2  -> Máxima distancia entre dos segmentos para ser
//                  considerados pertenecientes a la misma línea.

// Extracción de la primera línea de todas las obtenidas en la
// transformada de Hough.

CvPoint* linea = (CvPoint*)cvGetSeqElem(lineas,0);

// char* cvGetSeqElem(const CvSeq* seq, int index)
//    seq   -> Secuencia.
//    index -> Índice del elemento.

free(aux_v);
free(v_disp);

// BLOB ANALYSIS //

long num_obs;    // Número de obstáculos detectados.

// Carga del fichero.

MbufPut(MilImageRight, mapa_obs);


// Habilitación de la lista donde se almacenarán las características
// de los blobs.

MblobAllocFeatureList(MilSystem,&FeatureList); // Memoria para la lista.

// Habilitación de los diversos campos que contiene la lista de
// características de los blobs.

MblobSelectFeature(FeatureList, M_AREA); // Número de píxeles de los blobs.

MblobSelectFeature(FeatureList, M_BOX_X_MIN); // Coordenada x mínima.
MblobSelectFeature(FeatureList, M_BOX_Y_MIN); // Coordenada y mínima.
MblobSelectFeature(FeatureList, M_BOX_X_MAX); // Coordenada x máxima.
MblobSelectFeature(FeatureList, M_BOX_Y_MAX); // Coordenada y máxima.

// Establecimiento de memoria para el resultado de blobs.

MblobAllocResult(MilSystem,&BlobResult);

// Cálculo de los blobs.

MblobCalculate(MilImageRight, M_NULL, FeatureList, BlobResult);
```

```
// Eliminación de los blobs menores de M_AREA para eliminar ruido a la
// solución.

MblobSelect(BlobResult, M_EXCLUDE, M_AREA, M_LESS_OR_EQUAL, umbral_blob,
            M_NULL);

// Extracción del número total de blobs.

MblobGetNumber(BlobResult, &num_obs);

// Creación de estructuras dinámicas donde almacenar las coordenadas
// de los blobs.

long* x_min = (long*)malloc(num_obs*sizeof(long));
long* y_min = (long*)malloc(num_obs*sizeof(long));
long* x_max = (long*)malloc(num_obs*sizeof(long));
long* y_max = (long*)malloc(num_obs*sizeof(long));

// Extracción de las coordenadas de los blobs.
MblobGetResult(BlobResult, M_BOX_X_MIN+M_TYPE_LONG, x_min);
MblobGetResult(BlobResult, M_BOX_Y_MIN+M_TYPE_LONG, y_min);
MblobGetResult(BlobResult, M_BOX_X_MAX+M_TYPE_LONG, x_max);
MblobGetResult(BlobResult, M_BOX_Y_MAX+M_TYPE_LONG, y_max);

final=clock()-init;

free(x_min);
free(y_min);
free(x_max);
free(y_max);

free(disp_izda);
free(mapa_obs);
free(mapa_lib);

MbufFree(MilImageAux1);
MbufFree(MilImageAux2);
MbufFree(MilImageDisp);
MbufFree(MilImageLeft);
MbufFree(MilImageRight);

MblobFree(FeatureList);
MblobFree(BlobResult);

MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);

}

////////////////////////////////////////////////////////////////////////

//LAPLACIANA.
// Desde el punto de vista computacional, el cálculo de la Laplaciana
// de la Gaussiana de una imagen dada se reduce a evaluar cada píxel
// en función de sus vecinos mediante una máscara predeterminada (lg).

void laplaciana(unsigned char* im, int* lg, unsigned char* lp)
{
  // for(j) y for(i) recorren todos los píxeles de la imagen. Notar
  // que se excluye del cálculo un borde de 2 px alrededor de la imagen.
  // Sólo se van a evaluar con respecto a sus vecinos aquellos píxeles
  // que estén rodeados por, al menos, 2 píxeles en todas direcciones.
```

```
   // El área de sombra, en comparación con el tamaño total de la imagen,
   // es despreciable.

   for(int y=2; y<(tam_y-2); y++)
   {
      for(int x=2; x<(tam_x-2); x++)
      {

         // Se ha desarrollado el algoritmo para conseguir un ahorro de
         // tiempo de computo de entorno al 50%. De forma teórica se va
         // a trabajar sobre la versión reducida, siendo la versión
         // ampliada idéntica a ésta.
         // for(j) y for(i) recorren todos los píxeles de la máscara.

         // Para evitar la saturación del píxel sobre el que se trabaja,
         // el filtro se aplica con el coeficiente 80, consiguiendo así
         // mantener la ganancia a 1.

         // Para centrar la imange en el espectro de color medio, se le
         // suma a cada píxel 177 (la mitad del total, 254).

         lp[x+y*tam_x]=im[(x-2  )+(y-2)*tam_x]*lg[ 0]/80
                      +im[(x-1  )+(y-2)*tam_x]*lg[ 1]/80
                      +im[(x    )+(y-2)*tam_x]*lg[ 2]/80
                      +im[(x+1  )+(y-2)*tam_x]*lg[ 3]/80
                      +im[(x+2  )+(y-2)*tam_x]*lg[ 4]/80
                      +im[(x-2  )+(y-1)*tam_x]*lg[ 5]/80
                      +im[(x-1  )+(y-1)*tam_x]*lg[ 6]/80
                      +im[(x    )+(y-1)*tam_x]*lg[ 7]/80
                      +im[(x+1  )+(y-1)*tam_x]*lg[ 8]/80
                      + im[(x+2 )+(y-1)*tam_x]*lg[ 9]/80
                      +im[(x-2  )+(y  )*tam_x]*lg[10]/80
                      +im[(x-1  )+(y  )*tam_x]*lg[11]/80
                      +im[(x    )+(y  )*tam_x]*lg[12]/80
                      +im[(x+1  )+(y  )*tam_x]*lg[13]/80
                      + im[(x+2 )+(y  )*tam_x]*lg[14]/80
                      + im[(x-2 )+(y+1)*tam_x]*lg[15]/80
                      + im[(x-1 )+(y+1)*tam_x]*lg[16]/80
                      + im[(x   )+(y+1)*tam_x]*lg[17]/80
                      + im[(x+1 )+(y+1)*tam_x]*lg[18]/80
                      + im[(x+2 )+(y+1)*tam_x]*lg[19]/80
                      + im[(x-2 )+(y+2)*tam_x]*lg[20]/80
                      + im[(x-1 )+(y+2)*tam_x]*lg[20]/80
                      + im[(x   )+(y+2)*tam_x]*lg[22]/80
                      + im[(x+1 )+(y+2)*tam_x]*lg[23]/80
                      + im[(x+2 )+(y+2)*tam_x]*lg[24]/80+ 127;

      }
   }
}

////////////////////////////////////////////////////////////////////

//CÁLCULO DE LA DISPARIDAD.

// Este algoritmo calcula el mapa de disparidad de dos imágenes,
// optimizando el coste computacional y obteniendo, de forma simultánea,
// la disparidad de la imagen izquierda con respecto a la imagen derecha y
// la disparidad de la imagen derecha con respecto a la izquierda.
```

```
// El algoritmo se divide en dos partes diferenciadas: el cálculo de
// la primera columna y de las demás. Dentro de cada
// columna, a su vez, se haya en primer lugar la ventana superior y,
// a partir de ella, el resto.

// En la primera se obtiene, en primer lugar, la ventana superior,
// hayando todas las SAD que la componen. Las siguientes ventanas
//se calculan restando la contribución al coste de la fila que sólo
// pertenece a la antigua y sumándole la privativa de la nueva.

// Estos valores se almacenan en la matriz coste_fila. En cada fila
// de dicha matriz se conservan los costes resultantes al comparar
// una ventana de la imagen izquierda (y,x) con la correspondiente de
// la imagen derecha (y,x) y sucesivas, hasta (y,x+tam_coste).

// Para los primeros elementos de las demás columnas, procederemos
// fila a fila a partir del valor guardado en la matriz
// coste_fila. Para cada fila, restaremos el valor de SAD del píxel de
// la derecha, que pertenecía en exclusiva a la anterior
// ventana, sumándole el del de la izquierda, privativo de la nueva.
// Para este primer elemento, obtendremos el valor del coste
// de la ventana sumando todas sus filas.

// Para los demás elementos de la columna, partiremos del anterior,
// aprovechando el solapamiento que se produce entre dos ventanas
// correlativas. Los valores de cada fila de todas las ventanas están
// almacenados en la matriz coste_fila. Por tanto, basta con
// actualizar este valor a la columna siguiente, restándole el píxel
// izquierdo y sumándole el derecho.

// Además, para evitar el cálculo redundante de AD, al comienzo de la
// función se calculan todas las diferencias necesarias a lo largo de
// la misma, almacenándolas en el vector AD.

void disparidad(unsigned char* im_izda,unsigned char* im_dcha, unsigned
                char* disp_izda,unsigned char* disp_dcha)
{
  // Se definen las matrices de datos:
  //  - coste_fila almacena los resultados de comparar una columna de
  //     la imagen izquierda con la correspondiente de la derecha.
  //     A lo largo de cada fila se almacenan la comparación del píxel
  //     izquierdo (y,x) con el derecho (y,x) y posteriores, hasta
  //     (y,x+tam_coste).
  //     En cada fila de coste_fila se almacenan los valores
  //    correspondientes a esa misma fila de la imagen.
  //  - coste_dcha almacena los resultados de comparar la imagen
  //     derecha con la izquierda.
  //     Cada celda de la matriz guarda la menor AD encontrada para esa
  //     posición.
  //  - AD almacena las diferencias absolutas requeridas a lo largo
  //     del algoritmo.
  //     AD[k+y*tam_coste+x*tam_y*tam_coste]
  //     La matriz AD es un conjunto de submatrices, cada una de las
  //     cuales contiene la información de una columna. En cada una
  //     de estas submatrices se almacena, en sentido x, la AD entre
  //     el píxel izquierdo (y,x) y el derecho (y,x), y sucesivos, hasta
  //     (y,x+tam_coste). En sentido y, recorre todos los elementos de
  //     la columna.

  int* coste_fila=(int*)calloc(tam_coste*tam_y,sizeof(int));
  int* coste_dcha=(int*)calloc(tam_x*tam_y,sizeof(int));
```

```
int* AD=(int*)calloc(tam_coste*tam_x*tam_y,sizeof(int));

// Para posibilitar que la matriz coste_dcha guarde la menor AD
// encontrada para cada posición, se inicializa en el mayor valor
// posible.

int lim = 255*(2*rad_vent+1)*(2*rad_vent+1);
for(int d=0; d<(tam_x*tam_y); d++)
{ coste_dcha[d]=lim; }

// CÁLCULO DE LAS AD.

// A continuación se calculan todas las AD requeridas a lo largo del
// algoritmo.
//  - for(j) y for(i) recorren todos los píxeles de la matriz.
//  - for(k) permite comparar la ventana izquierda (y,x) con la
//     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

for (int j=0; j<tam_y; j++)
{
  for (int i=0; i<tam_x; i++)
  {
    for (int k=0; k<tam_coste; k++)
    {
      AD[k+j*tam_coste+i*tam_y*tam_coste]=
          AD[k+j*tam_coste+i*tam_y*tam_coste]
         +abs(im_izda[i+tam_x*j]-im_dcha[k+i+tam_x*j]);
    }
  }
}

// Primera columna · Primer elemento.

// coste almacena las SAD entre el píxel izquierdo (y,x) y el
// derecho (y,x), y sucesivos, hasta (y,x+tam_coste).
// Este valor se inicializará para cada columna (a partir del
// primer elemento de cada columna, se calculan los demás).

int* coste=(int*) calloc(tam_coste, sizeof(int));

// A continuación se calculan las SAD del primer elemento de la
// primera columna de la imagen izquierda.
//  - for(k) permite comparar la ventana izquierda (y,x) con la
//     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
//  - for(j) y for(i) recorren todos los píxeles de la ventana.
//El resultado de acumular todas las AD de una fila de la ventana
// de búsqueda se almacena en coste_fila.
//El resultado de la ventana de búsqueda completa se almacena en coste.

  for(int k=0; k<tam_coste; k++)
  {
    for(int j=-rad_vent; j<=rad_vent; j++)
    {
      for(int i=-rad_vent; i<=rad_vent; i++)
      {
        coste_fila[k+(rad_vent+j)*tam_coste]=
            coste_fila[k+(rad_vent+j)*tam_coste]
           +AD[k+(rad_vent+j)*tam_coste+(rad_vent+i)*tam_y*tam_coste];
      }
      coste[k]=coste[k]+coste_fila[k+(rad_vent+j)*tam_coste];
    }
```

```
    // En disp_izda se almacena la menor disparidad izquierda encontrada
    // hasta el momento para cada píxel. Este valor se compara con cada
    // SAD a medida que se calculan, guardando el nuevo valor en caso de
    // ser menor.

    if(coste[k]<coste[disp_izda[rad_vent+(tam_x*rad_vent)]])
      { disp_izda[rad_vent+(tam_x*rad_vent)]=k; }

    // En coste_dcha se almacena el menor valor de SAD encontrado
    // hasta el momento para cada píxel. Este valor se compara con
    // cada SAD a medida que se calculan, guardando el nuevo valor en
    // caso de ser menor. En disp_dcha se almacena, de forma simultánea,
    // la disparidad a la que corresponde el valor guardado en coste_dcha.

    if(coste[k]<coste_dcha[k+rad_vent+(tam_x*rad_vent)])
    {
      disp_dcha[k+rad_vent+(tam_x*rad_vent)]=k;
      coste_dcha[k+rad_vent+(tam_x*rad_vent)]=coste[k];
    }
  }

  // Primera columna · Demás elementos.

  // A continuación se calculan las SAD de los demás elementos de la
  // primera columna de la imagen izquierda.
  //   - for(y) permite recorrer la columna en sentido descendente.
  //   - for(k) permite comparar la ventana izquierda (y,x) con la
  //        ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
  //   - for(i) recorre el borde inferior de la ventana de búsqueda,
  //        calculando la nueva fila implicada.

  for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
  {
    for(int k=0; k<tam_coste; k++)
    {
      for(int i=-rad_vent; i<=rad_vent; i++)
      {
      // Al valor de coste almacenado para la ventana precedente, se
      // le resta el valor de la fila superior (calculado previamente
      // y almacenado en la matriz coste_fila) y se le suma el valor
      // de la fila inferior (recién calculado).

        coste_fila[k+(y+rad_vent)*tam_coste]=
            coste_fila[k+(y+rad_vent)*tam_coste]
          +AD[k+(y+rad_vent)*tam_coste+(rad_vent+i)*tam_y*tam_coste];
      }

      coste[k]=coste[k]+coste_fila[k+(y+rad_vent)*tam_coste]
              -coste_fila[k+(y-(rad_vent+1))*tam_coste];

      if(coste[k]<coste[disp_izda[rad_vent+(tam_x*y)]])
        { disp_izda[rad_vent+(tam_x*y)]=k; }

      if(coste[k]<coste_dcha[k+rad_vent+(tam_x*y)])
      {
        disp_dcha[k+rad_vent+(tam_x*y)]=k;
        coste_dcha[k+rad_vent+(tam_x*y)]=coste[k];
      }
    }
  }
```

```
free(coste);

// for(x) recorre todas las columnas de la imagen de izquierda a derecha.

for (int x=(rad_vent+1); x<(tam_x-tam_coste); x++)
{

// Demás columnas · Primer elemento.

int* coste=(int*) calloc(tam_coste, sizeof(int));

// A continuación se calculan las SAD del primer elemento de las
// demás columnas.
//   - for(k) permite comparar la ventana izquierda (y,x) con la
//        ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
//   - for(j) recorre los bordes laterales de la ventana de
//        búsqueda. Al valor almacenado de cada fila(correspondiente a la
//        columna anexa) se le resta la AD del píxel izquierdo,
//        sumándole la del derecho para obtener el valor de la fila de la
//        nueva columna.

for(int k=0; k<tam_coste; k++)
{
  for(int j=-rad_vent; j<=rad_vent; j++)
  {
    coste_fila[k+(rad_vent+j)*tam_coste]=
       coste_fila[k+(rad_vent+j)*tam_coste]
      +AD[k+(rad_vent+j)*tam_coste+(x+ rad_vent)*tam_y*tam_coste]
      -AD[k+(rad_vent+j)*tam_coste+(x(rad_vent+1))*tam_y*tam_coste];

    coste[k]=coste[k]+coste_fila[k+(rad_vent+j)*tam_coste];
  }

  if(coste[k]<coste[disp_izda[x+(tam_x*rad_vent)]])
    { disp_izda[x+(tam_x*rad_vent)]=k; }

  if(coste[k]<coste_dcha[k+x+(tam_x*rad_vent)])
  {
    disp_dcha[k+x+(tam_x*rad_vent)]=k;
    coste_dcha[k+x+(tam_x*rad_vent)]=coste[k];
  }
}

// Demás columnas · Demás elementos.

// A continuación se calculan las SAD de los demás elementos.
//   - for(y) permite recorrer la columna en sentido descendente.
//   - for(k) permite comparar la ventana izquierda (y,x) con la
//        ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
{
  for(int k=0; k<tam_coste; k++)
  {
  // A partir del valor de la columna anexa, se actualiza el valor
  // de la fila inferior de la cada ventana.
```

```
        coste_fila[k+(y+rad_vent)*tam_coste]=
            coste_fila[k+(y+rad_vent)*tam_coste]
           +AD[k+(y+rad_vent)*tam_coste+(x+ rad_vent)*tam_y*tam_coste]
           -AD[k+(y+rad_vent)*tam_coste+(x-(rad_vent+1))*tam_y*tam_coste];

        coste[k]=coste[k]+
            coste_fila[k+(y+rad_vent)*tam_coste]
           -coste_fila[k+(y-(rad_vent+1))*tam_coste];

        if(coste[k]<coste[disp_izda[x+(tam_x*y)]])
           { disp_izda[x+(tam_x*y)]=k; }

        if(coste[k]<coste_dcha[k+x+(tam_x*y)])
        {
           disp_dcha[k+x+(tam_x*y)]=k;
           coste_dcha[k+x+(tam_x*y)]=coste[k];
        }
      }
   }

   free(coste);
}
   free(coste_dcha);
   free(coste_fila);
   free(AD);
}

////////////////////////////////////////////////////////////////////

//U_DISPARITY.
//La función u_disparity calcula el histograma, columna a columna, del
// mapa de disparidad. Como resultado se obtiene la imagen u_disp.

void u_disparity(unsigned char* disp_izda, unsigned char* u_disp)
{
   // for(y) y for(x) recorren todos los píxeles del mapa de disparidad.

   for(int y=0; y<tam_y; y++)
   {
      for(int x=0; x<tam_x; x++)
      {
         // La matriz u_disparity tiene un tamaño (tam_coste, tam_x):
         // - La coordenada x indica la columna de la imagen a la que
         //   corresponde esa columna del histograma.
         // - La coordenada y indica el valor de la disparidad que
         //   representa dicho píxel. Para d=0, y=0; para d=dmáx, y=dmáx.
         // - El valor del píxel representa la densidad de dicha
         //   disparidad (d=y) en esa columna (x).
         // Se comprueba que el valor del píxel sea menor que 255 (el
         // máximo) para evitar desbordamientos.

         if(u_disp[x+disp_izda[x+y*tam_x]*tam_x]!=255)
            {u_disp[x+disp_izda[x+y*tam_x]*tam_x]++;}
      }
   }

   // Para evitar datos erróneos a la hora de detectar los obstáculos
   // posteriormente, la fila superior, que siempre mostrará
   // valores elevados de densidad para el color negro (que se asume
   // como color de fondo), se anula.
```

```
   for (int k=0; k<tam_x; k++)
   { u_disp[k]=0; }

   // Se binarizan los valores de u_disp en función de umbral_u.

   for(int k=tam_x; k<(tam_x*tam_coste); k++)
   {
      if(u_disp[k]<=umbral_u)
      { u_disp[k]=0; }
      else
      { u_disp[k]=255; }
   }
}

////////////////////////////////////////////////////////////////////

// MAPA OBSTÁCULO Y MAPA LIBRE.
// El mapa obstáculo se obtiene mediante la extracción del mapa de
// disparidad (disp_izda) los píxeles marcados como obstáculos por la
// matriz u_disparity. El resto de los píxeles, que se asume que no
// forman parte de ningún obstáculo, conforman el mapa libre (de
// obstáculos).

void mapas(unsigned char* disp_izda,unsigned char* u_disp,unsigned
           char* mapa_obs,unsigned char* mapa_lib)
{
   // for(x) y for(y) recorren todos los píxeles de la u_disparity.

   for(int x=0; x<tam_x; x++)
   {
      for(int y=0; y<tam_coste; y++)
      {
         // Se localizan los píxeles distintos de cero en el histograma
         // (u_disparity), que es donde se asume que están los obstáculos.

         if(u_disp[x+y*tam_x]!=0)
         {
            // Para cada píxel distinto de cero, el bucle for(y_disp)
            // recorre toda la columna correspondiente de la imagen.

            for(int y_disp=0; y_disp<tam_y; y_disp++)
            {
               // Los píxeles de la imagen cuyo valor coincida con los
               // marcados como obstáculos por el histograma, se cargan
               // en el mapa obstáculo (mapa_obs).

               if(disp_izda[x+y_disp*tam_x]==y)
               {
                  mapa_obs[x+y_disp*tam_x]=disp_izda[x+y_disp*tam_x];
               }
            }
         }
      }
   }

   // for(k) recorre todos los píxeles de la imagen. Se cargan en el
   // mapa libre todos aquellos valores de la disparidad que no forman
   // parte del mapa obstáculo.

   for (int k=0; k<(tam_x*tam_y); k++)
   {
```

```
      if(mapa_obs[k]==0)
        {mapa_lib[k]=disp_izda[k];}
   }

   // Se binarizan los valores del mapa obstáculo en función de umbral_obs.

   for(int k=0; k<(tam_x*tam_y); k++)
   {
      if(mapa_obs[k]<umbral_obs)
        { mapa_obs[k]=0; }
      else
        { mapa_obs[k]=255; }
   }
}

////////////////////////////////////////////////////////////////////////

// V_DISPARITY.
// La función v_disparity calcula el histograma, fila a fila, del
// mapa de disparidad. Como resultado se obtiene la imagen v_disp.

void v_disparity(unsigned char* disp_izda, unsigned char* v_disp)
{
   // for(y) y for(x) recorren todos los píxeles del mapa de disparidad.

   for(int y=0; y<tam_y; y++)
   {
      for(int x=0; x<tam_x; x++)
      {
        // La matriz v_disparity tiene un tamaño (tam_y, tam_coste):
        // - La coordenada y indica la fila de la imagen a la que
        //   corresponde esa fila del histograma.
        // - La coordenada x indica el valor de la disparidad que
        //   representa dicho píxel. Para d=0, x=0; para d=dmáx, x=dmáx.
        // - El valor del píxel representa la densidad de dicha
        //   disparidad (d=x) en esa columna (y).
        // Se comprueba que el valor del píxel sea menor que 255 (el
        // máximo) para evitar desbordamientos.

        if (v_disp[y*tam_coste+disp_izda[x+y*tam_x]]!=255)
          {v_disp[y*tam_coste+disp_izda[x+y*tam_x]]++;}
      }
   }

   // Para evitar datos erróneos a la hora de detectar los obstáculos
   // posteriormente, la columna derecha, que siempre mostrará
   // valores elevados de densidad para el color negro (que se asume
   // como color de fondo), se va a anular.

   for (int k=0; k<tam_y; k++)
   { v_disp[k*tam_coste]=0; }

   // Se umbralizan los valores de v_disp en función de umbral_v.

   for(int k=tam_x; k<(tam_coste*tam_y); k++)
   {
      if(v_disp[k]<=umbral_v)
        { v_disp[k]=0; }
   }
}
```

# ANEXO 2 · EVOLUCIÓN DEL ALGORITMO DEL CÁLCULO DE LA DISPARIDAD

En esta sección se adjuntan los diversos códigos resultado de la evolución del algoritmo de cálculo de la disparidad.

En todos los casos, las imágenes originales son cargadas a formato matriz en el programa principal. Así mismo, las imágenes de partida de la función, `im_izda`, `im_dcha`, han sido tratadas previamente, aplicándoles la Laplaciana de la Gaussiana.

De igual modo, las imágenes destino, `disp` o `disp_izda y disp_dcha` en función del algoritmo, han sido definidas en el programa principal como:

```
unsigned char* disp = (unsigned char*)calloc(tam_x*tam_y,
  sizeof(unsigned char));
```

Cargándose en la función disparidad los resultados calculados.

Para tener una visión más clara sobre la integración del algoritmo del cálculo de la disparidad en el programa de detección y localización de obstáculos, se sugiere consultar el Anexo 1, donde aparece el código completo del mismo. En la llamada a la función disparidad sería posible situar cualquiera de los códigos siguientes, con la única excepción del primero, que únicamente tiene una matriz como resultado, debiéndose en este caso anular el cálculo del Crosschecking al carecer de sentido.

## VERSIÓN 1 · CÁLCULO DEL MAPA DE DISPARIDAD

```
////////////////////////////////////////////////////////////////////

VERSIÓN 1 · CÁLCULO DEL MAPA DE DISPARIDAD.
El algoritmo calcula el mapa de disparidad.

Ésto se consigue mediante tres grupos de bucles for anidados:
 - El bucle interno (compuesto por for(i) y for(j))recorre todos los
      elementos de una ventana (2*rad_vent+1)^2 para el cálculo de la
      SAD.
 - El segundo bucle (compuesto por for(k)) permite comparar la ventana
      izquierda (y,x) con la ventana derecha(y,x)y posteriores, hasta
      (y,x+tam_coste).
 - El bucle externo (compuesto por for(y) y for(x)) recorre todos los
      píxeles de la imagen izquierda. Desde la ventana superior
      izquierda, se desplaza en cada columna de arriba a abajo, y de
      izquierda a derecha en las filas.

////////////////////////////////////////////////////////////////////

void disparidad(unsigned char* im_izda, unsigned char* im_dcha,
                unsigned char* disp)
{
  // for(y) y for(x) recorren todos los píxeles de la imagen izquierda

  for (int x=rad_vent; x<(tam_x-rad_vent); x++)
  {
    for (int y=rad_vent; y<(tam_y-rad_vent); y++)
    {
      // coste almacena las SAD entre el píxel izquierdo (y,x) y el
      // derecho (y,x), y sucesivos, hasta (y,x+tam_coste).
      // Este valor se inicializará para cada ventana.

      int* coste=(int*) calloc(tam_coste, sizeof(int));

      // for(k) permite comparar la ventana izquierda (y,x) con la
      // ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

      for(int k=0; k<tam_coste; k++)
      {
        // for(j) y for(i) recorren todos los píxeles de la ventana
        // de búsqueda.

        for(int j=-rad_vent; j<=rad_vent; j++)
        {
          for(int i=-rad_vent; i<=rad_vent; i++)
          {
            coste[k]=coste[k]+abs(im_izda[(x+i)+tam_x*(y+j)]-
                                  im_dcha[(k+x+i)+tam_x*(y+j)]);
          }
        }
        // En disp se almacena la menor disparidad izquierda
        // encontrada hasta el momento para cada píxel. Este valor se
        // compara con cada SAD a medida que se calculan, guardando
        // el nuevo valor en caso de ser menor.

        if(coste[k]<coste[disp[x+(tam_x*y)]])
          { disp[x+(tam_x*y)]=k; }
      }
```

```
        free(coste);
      }
    }
}
```

## VERSIÓN 2 · CÁLCULO DEL MAPA DE DISPARIDAD A PARTIR DEL PRIMER ELEMENTO DE CADA COLUMNA

```
////////////////////////////////////////////////////////////////////////

VERSIÓN 2 · CÁLCULO DEL MAPA DE DISPARIDAD A PARTIR DEL PRIMER
ELEMENTO DE CADA COLUMNA.
Este algoritmo es una evolución de la Versión 1 cuyo objetivo es
reducir el coste computacional. El cálculo de las SAD de las ventanas
cada columna se lleva a cabo en dos pasos.

En primer lugar se calcula la SAD de la ventana superior de cada
columna, recorriendo todos sus píxeles.

A partir de este valor, se calcula el de las ventanas siguientes
restándole las AD de los píxeles de la fila privativa de la anterior
ventana de búsqueda, y sumándole los de la nueva fila.



El bucle externo (for(x)) recorre la imagen columna a columna.
  1.- Mediante los bucles anidados (for(j) y for(i)) se calcula la SAD
      de la ventana superior de cada columna.
  2.- Mediante el bucle for(y) se recorren los demás elementos de la
      columna de arriba a abajo. Para cada uno de ellos, al coste de
      la anterior fila se le resta la contribución de de los píxeles
      que sólo pertenecen a la antigua ventana, sumándole los
      privativos de la nueva. Ello se consigue mediante el bucle
      for(i), que recorre la ventana de izquierda a derecha.
En ambos casos, el bucle for(k) permite comparar la ventana izquierda
(y,x) con la ventana derecha(y,x) y posteriores, hasta
(y,x+tam_coste).

////////////////////////////////////////////////////////////////////////

void disparidad(unsigned char* im_izda, unsigned char* im_dcha,
                unsigned char* disp)
{
  // for(x) recorre todas las columnas de la imagen de izquierda a
  // derecha.

  for (int x=rad_vent; x<(tam_x-tam_coste); x++)
  {

//Todas las columnas · Primer elemento.

    // coste almacena las SAD entre el píxel izquierdo (y,x) y el
    // derecho (y,x), y sucesivos, hasta (y,x+tam_coste).
    // Este valor se inicializará para cada columna.

    int* coste=(int*) calloc(tam_coste, sizeof(int));

      // for(k) permite comparar la ventana izquierda (y,x) con la
      // ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

      for(int k=0; k<tam_coste; k++)
      {
        // for(j) y for(i) recorren todos los píxeles de la ventana
        // de búsqueda.
```

```
          for(int j=-rad_vent; j<=rad_vent; j++)
          {
            for(int i=-rad_vent; i<=rad_vent; i++)
            {
              coste[k]=coste[k]+abs(im_izda[(x+i)+tam_x*(rad_vent+j)]-
                                    im_dcha[(k+x+i)+tam_x*(rad_vent+j)]);
            }
          }

          // En disp se almacena la menor disparidad izquierda
          // encontrada hasta el momento para cada píxel. Este valor se
          // compara con cada SAD a medida que se calculan, guardando
          // el nuevo valor en caso de ser menor.

          if(coste[k]<coste[disp[x+(tam_x*rad_vent)]])
            { disp[x+(tam_x*rad_vent)]=k; }
        }

//Todas las columnas · Demás elementos.

     // A continuación se calculan las SAD de los demás elementos cada
     // columna.
     //  - for(y) permite recorrer la columna en sentido descendente.
     //  - for(k) permite comparar la ventana izquierda (y,x) con la
     //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
     //  - for(i) recorre el borde superior e inferior de la ventana de
     //     búsqueda.

     for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
     {
       for(int k=0; k<tam_coste; k++)
       {
         for(int i=-rad_vent; i<=rad_vent; i++)
         {
           // Al valor de coste almacenado para la ventana precedente,
           // se le resta el valor de la fila superior (que sólo
           // pertenece a la anterior ventana) y se le suma el valor
           // de la fila inferior (privativa de la nueva).

           coste[k]=coste[k]+abs(im_izda[(x+i)+tam_x*(y+(rad_vent))]-
                                 im_dcha[(k+x+i)+tam_x*(y+(rad_vent))])-
                             abs(im_izda[(x+i)+tam_x*(y-(rad_vent+1))]-
                                 im_dcha[(k+x+i)+tam_x*(y-(rad_vent+1))]);
         }

         if(coste[k]<coste[disp[x+(tam_x*y)]])
           { disp[x+(tam_x*y)]=k; }
       }
     }

     free(coste);
   }
}
```

## VERSIÓN 3 · CÁLCULO DE LAS DISPARIDADES IZQUIERDA Y DERECHA DE FORMA SIMULTÁNEA

```
////////////////////////////////////////////////////////////////////////

VERSIÓN 3 · CÁLCULO DE LAS DISPARIDADES IZQUIERDA Y DERECHA DE FORMA
SIMULTÁNEA.
El algoritmo es una evolución de la Versión 2 cuyo objetivo es
calcular, de manera simultánea, los mapas de disparidad izquierdo y
derecho. Como se demostró en el Capítulo 4 de la memoria, todas las
SAD necesarias para el cálculo de la disparidad derecha han sido
previamente obtenidas durante el cálculo de la disparidad izquierda.

Durante el cálculo de la disparidad izquierda (que no varía con
respecto a la versión 2), los costes resultantes de comparar la
ventana izquierda (y,x) con la ventana derecha(y,x) y sucesivas, hasta
(y,x+tam_coste), se almacenan en el vector coste, de tamaño tam_coste.
En el caso de la disparidad derecha, al calcularse las SAD de forma
desordenada, se requiere una matriz de las mismas dimensiones que las
imágenes para almacenar el valor del menor coste encontrado para cada
píxel.

////////////////////////////////////////////////////////////////////////

void disparidad(unsigned char* im_izda, unsigned char* im_dcha,
                unsigned char* disp_izda, unsigned char* disp_dcha)
{
  // coste almacena las SAD entre el píxel izquierdo (y,x) y el
  // derecho (y,x), y sucesivos, hasta (y,x+tam_coste).
  // Este valor se inicializará para cada columna.

  int* coste_dcha=(int*)calloc(tam_x*tam_y, sizeof(int));

  // Para posibilitar que la matriz coste_dcha guarde la menor AD
  // encontrada para cada posición, se inicializa en el mayor valor
  // posible.

  int lim = 255*(2*rad_vent+1)*(2*rad_vent+1);
  for(int d=0; d<(tam_x*tam_y); d++)
  { coste_dcha[d]=lim; }

  // for(x) recorre todas las columnas de la imagen de izquierda a
  // derecha.

  for (int x=rad_vent; x<(tam_x-tam_coste); x++)
  {

// Todas las columnas · Primer elemento.

    int* coste=(int*) calloc(tam_coste, sizeof(int));

    // for(k) permite comparar la ventana izquierda (y,x) con la
    // ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

    for(int k=0; k<tam_coste; k++)
    {
      // for(j) y for(i) recorren todos los píxeles de la ventana de
      // búsqueda.
```

```
        for(int j=-rad_vent; j<=rad_vent; j++)
        {
          for(int i=-rad_vent; i<=rad_vent; i++)
          {
            coste[k]=coste[k]+abs(im_izda[(x+i)+tam_x*(rad_vent+j)]-
                                  im_dcha[(k+x+i)+tam_x*(rad_vent+j)]);
          }
        }

        // En disp se almacena la menor disparidad izquierda encontrada
        // hasta el momento para cada píxel. Este valor se compara con
        // cada SAD a medida que se calculan, guardando el nuevo valor
        // en caso de ser menor.

        if(coste[k]<coste[disp_izda[x+(tam_x*rad_vent)]])
          { disp_izda[x+(tam_x*rad_vent)]=k; }

        // En coste_dcha se almacena el menor valor de SAD encontrado
        // hasta el momento para cada píxel. Este valor se compara con
        // cada SAD a medida que se calculan, guardando el nuevo valor
        // en caso de ser menor. En disp_dcha se almacena, de forma
        // simultánea, la disparidad a la que corresponde el valor
        // guardado en coste_dcha.

        if(coste[k]<coste_dcha[k+x+(tam_x*rad_vent)])
        {
          disp_dcha[k+x+(tam_x*rad_vent)]=k;
          coste_dcha[k+x+(tam_x*rad_vent)]=coste[k];
        }
      }

// Todas las columnas · Demás elementos.

      // A continuación se calculan las SAD de los demás elementos del
      // resto de las columnas de la imagen izquierda.
      //  - for(y) permite recorrer la columna en sentido descendente.
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
      //  - for(i) recorre el borde inferior de la ventana de búsqueda,
      //     calculando la nueva fila implicada.

      for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
      {
        for(int k=0; k<tam_coste; k++)
        {
          for(int i=-rad_vent; i<=rad_vent; i++)
          {
            // Al valor de coste almacenado para la ventana precedente,
            // se le resta el valor de la fila superior (que sólo
            // pertenece a la anterior ventana) y se le suma el valor
            // de la fila inferior (privativa de la nueva).

            coste[k]=coste[k]-abs(im_izda[(x+i)+tam_x*(y-(rad_vent+1))]-
                                  im_dcha[(k+x+i)+tam_x*(y-(rad_vent+1))])+
                             abs(im_izda[(x+i)+tam_x*(y+rad_vent)]-
                                  im_dcha[(k+x+i)+tam_x*(y+rad_vent)]);
          }

          if(coste[k]<coste[disp_izda[x+(tam_x*y)]])
            { disp_izda[x+(tam_x*y)]=k; }
```

```
          if(coste[k]<coste_dcha[k+x+(tam_x*y)])
          {
            disp_dcha[k+x+(tam_x*y)]=k;
            coste_dcha[k+x+(tam_x*y)]=coste[k];
          }
        }
      }

      free(coste);
    }
}
```

## VERSIÓN 4 · CÁLCULO DEL MAPA DE DISPARIDAD MEDIANTE LA DESCOMPOSICIÓN DE LA VENTANA EN FILAS

```
////////////////////////////////////////////////////////////////////

VERSIÓN 4 · CÁLCULO DEL MAPA DE DISPARIDAD MEDIANTE LA DESCOMPOSICIÓN
DE LA VENTANA EN FILAS.
Este algoritmo es una evolución de la Versión 3 cuyo objetivo es
reducir el coste computacional, manteniendo el cálculo simultáneo de
las dos disparidades.

El algoritmo se divide en dos partes diferenciadas: el cálculo de la
primera columna y de las demás. Dentro de cada columna, a su vez, se
haya en primer lugar la ventana superior y, a partir de ella, el
resto.

La primera columna se calcula de forma análoga a la expuesta en la
Versión 2. En primer lugar se obtiene la ventana superior, hayando
todas las SAD que la componen. Las siguientes ventanas se calculan
restando la contribución al coste de la fila que sólo pertenece a la
antigua y sumándole la privativa de la nueva.

La única diferencia con respecto al anterior desarrollo cosiste en que
en este caso los valores no se almacenan directamente en el vector
coste, sino en la matriz coste_fila. En cada fila de la matriz se
conservan los costes resultantes al comparar una ventana de la imagen
izquierda (y,x) con la correspondiente de la imagen derecha (y,x) y
sucesivas, hasta (y,x+tam_coste).

Para los primeros elementos de las demás columnas, procederemos fila a
fila a partir del valor guardado en la matriz coste_fila. Para cada
fila, restaremos el valor de SAD del píxel de la derecha, que
pertenecía en exclusiva a la anterior ventana, sumándole el del de la
izquierda, privativo de la nueva. Para este primer elemento,
obtendremos el valor del coste de la ventana sumando todas sus filas.

Para los demás elementos de la columna, partiremos del anterior y
procederemos de forma análoga a la de la Versión 2. Es decir,
aprovechando el solapamiento que se produce entre dos ventanas
correlativas. La mejora con respecto al anterior algoritmo se basa en
que los valores de cada fila de todas las ventanas están almacenados
en la matriz coste_fila. Por tanto, basta con actualizar este valor a
la columna siguiente, restándole el píxel izquierdo y sumándole el
derecho.

Además, para evitar el cálculo redundante de AD, se calculan todas las
diferencias necesarias al comienzo de la función, almacenándose en el
vector AD.

////////////////////////////////////////////////////////////////////

void disparidad(unsigned char* im_izda, unsigned char* im_dcha,
                unsigned char* disp_izda, unsigned char* disp_dcha)
{
  // Se definen las matrices de datos:
  //  - coste_fila almacena los resultados de comparar una columna de
  //       la imagen izquierda con la correspondiente de la derecha.
  //     A lo largo de cada fila se almacenan la comparación del píxel
  //       izquierdo (y,x) con el derecho (y,x) y posteriores, hasta
  //       (y,x+tam_coste).
```

```
   //     En cada fila de coste_fila se almacenan los valores
   //       correspondientes a esa misma fila de la imagen.
   //  - coste_dcha almacena los resultados de comparar la imagen
   //       derecha con la izquierda.
   //     Cada celda de la matriz guarda la menor AD encontrada para
   //       esa posición.
   //  - AD almacena las diferencias absolutas requeridas a lo largo
   //       del algoritmo.
   //     AD[k+y*tam_coste+x*tam_y*tam_coste]
   //     La matriz AD es un conjunto de submatrices, cada una de las
   //       cuales contiene la información de una columna. En cada una
   //       de estas submatrices se almacena, en sentido x, la AD entre
   //       el píxel izquierdo (y,x) y el derecho (y,x), y sucesivos,
   //       hasta (y,x+tam_coste). En sentido y, recorre todos los
   //       elementos de la columna.

   int* coste_fila=(int*)calloc(tam_coste*tam_y,sizeof(int));
   int* coste_dcha=(int*)calloc(tam_x*tam_y,sizeof(int));
   int* AD=(int*)calloc(tam_coste*tam_x*tam_y,sizeof(int));

   // Para posibilitar que la matriz coste_dcha guarde la menor AD
   // encontrada para cada posición, se inicializa en el mayor valor
   // posible.

   int lim = 255*(2*rad_vent+1)*(2*rad_vent+1);
   for(int d=0; d<(tam_x*tam_y); d++)
   { coste_dcha[d]=lim; }

   // CÁLCULO DE LAS AD.

   // A continuación se calculan todas las AD requeridas a lo largo del
   // algoritmo.
   //  - for(j) y for(i) recorren todos los píxeles de la matriz.
   //  - for(k) permite comparar la ventana izquierda (y,x) con la
   //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

   for (int j=0; j<tam_y; j++)
   {
      for (int i=0; i<tam_x; i++)
      {
         for (int k=0; k<tam_coste; k++)
         {
            AD[k+j*tam_coste+i*tam_y*tam_coste]=
                 AD[k+j*tam_coste+i*tam_y*tam_coste]+
                 abs(im_izda[i+tam_x*j]-im_dcha[k+i+tam_x*j]);
         }
      }
   }

// Primera columna · Primer elemento.

     // coste almacena las SAD entre el píxel izquierdo (y,x) y el
     // derecho (y,x), y sucesivos, hasta (y,x+tam_coste).
     // Este valor se inicializará para cada columna (a partir del
     // primer elemento de cada columna, se calculan los demás).

     int* coste=(int*) calloc(tam_coste, sizeof(int));

     // A continuación se calculan las SAD del primer elemento de la
     // primera columna de la imagen izquierda.
```

```
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
      //  - for(j) y for(i) recorren todos los píxeles de la ventana.
      // El resultado de acumular todas las AD de una fila de la ventana
      // de búsqueda se almacena en coste_fila.
      // El resultado de la ventana de búsqueda completa se almacena en
      // coste.

      for(int k=0; k<tam_coste; k++)
      {
        for(int j=-rad_vent; j<=rad_vent; j++)
        {
          for(int i=-rad_vent; i<=rad_vent; i++)
          {
            coste_fila[k+(rad_vent+j)*tam_coste]=
                 coste_fila[k+(rad_vent+j)*tam_coste]+
                 AD[k+(rad_vent+j)*tam_coste+(rad_vent+i)*tam_y*tam_coste];
          }
          coste[k]=coste[k]+coste_fila[k+(rad_vent+j)*tam_coste];
        }

        // En disp_izda se almacena la menor disparidad izquierda
        // encontrada hasta el momento para cada píxel. Este valor se
        // compara con cada SAD a medida que se calculan, guardando el
        // nuevo valor en caso de ser menor.

        if(coste[k]<coste[disp_izda[rad_vent+(tam_x*rad_vent)]])
          { disp_izda[rad_vent+(tam_x*rad_vent)]=k; }

        // En coste_dcha se almacena el menor valor de SAD encontrado
        // hasta el momento para cada píxel. Este valor se compara con
        // cada SAD a medida que se calculan, guardando el nuevo valor
        // en caso de ser menor. En disp_dcha se almacena, de forma
        // simultánea, la disparidad a la que corresponde el valor
        // guardado en coste_dcha.

        if(coste[k]<coste_dcha[k+rad_vent+(tam_x*rad_vent)])
          {disp_dcha[k+rad_vent+(tam_x*rad_vent)]=k;
          coste_dcha[k+rad_vent+(tam_x*rad_vent)]=coste[k];
          }
      }

// Primera columna · Demás elementos.

      // A continuación se calculan las SAD de los demás elementos de la
      // primera columna de la imagen izquierda.
      //  - for(y) permite recorrer la columna en sentido descendente.
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
      //  - for(i) recorre el borde inferior de la ventana de búsqueda,
      //     calculando la nueva fila implicada.

      for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
      {
        for(int k=0; k<tam_coste; k++)
        {
          for(int i=-rad_vent; i<=rad_vent; i++)
          {
            // Al valor de coste almacenado para la ventana precedente,
            // se le resta el valor de la fila superior (calculado
```

```
            // previamente y almacenado en la matriz coste_fila) y se
            // le suma el valor de la fila inferior (recién calculado).

            coste_fila[k+(y+rad_vent)*tam_coste]=
                 coste_fila[k+(y+rad_vent)*tam_coste]+
                 AD[k+(y+rad_vent)*tam_coste+(rad_vent+i)*tam_y*tam_coste];
          }

          coste[k]=coste[k]+coste_fila[k+(y+rad_vent)*tam_coste]-
               coste_fila[k+(y-(rad_vent+1))*tam_coste];

          if(coste[k]<coste[disp_izda[rad_vent+(tam_x*y)]])
            { disp_izda[rad_vent+(tam_x*y)]=k; }

          if(coste[k]<coste_dcha[k+rad_vent+(tam_x*y)])
          {
            disp_dcha[k+rad_vent+(tam_x*y)]=k;
            coste_dcha[k+rad_vent+(tam_x*y)]=coste[k];
          }
        }
      }

      free(coste);

    // for(x) recorre todas las columnas de la imagen de izquierda a
    // derecha.

    for (int x=(rad_vent+1); x<(tam_x-tam_coste); x++)
    {

// Demás columnas · Primer elemento.

      int* coste=(int*) calloc(tam_coste, sizeof(int));

      // A continuación se calculan las SAD del primer elemento de las
      // demás columnas.
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
      //  - for(j) recorre los bordes laterales de la ventana de
      //     búsqueda. Al valor almacenado de cada fila(correspondiente
      //     a la columna anexa) se le resta la AD del píxel izquierdo,
      //     sumándole la del derecho para obtener el valor de la fila
      //     de la nueva columna.

      for(int k=0; k<tam_coste; k++)
      {
        for(int j=-rad_vent; j<=rad_vent; j++)
        {
          coste_fila[k+(rad_vent+j)*tam_coste]=
               coste_fila[k+(rad_vent+j)*tam_coste]+
               AD[k+(rad_vent+j)*tam_coste+(x+rad_vent)*tam_y*tam_coste]-
               AD[k+(rad_vent+j)*tam_coste+(x-(rad_vent+1))*tam_y*tam_coste];

          coste[k]=coste[k]+coste_fila[k+(rad_vent+j)*tam_coste];
        }

        if(coste[k]<coste[disp_izda[x+(tam_x*rad_vent)]])
          { disp_izda[x+(tam_x*rad_vent)]=k; }

        if(coste[k]<coste_dcha[k+x+(tam_x*rad_vent)])
        {
```

```
          disp_dcha[k+x+(tam_x*rad_vent)]=k;
          coste_dcha[k+x+(tam_x*rad_vent)]=coste[k];
        }
      }

// Demás columnas · Demás elementos.

      // A continuación se calculan las SAD de los demás elementos.
      //  - for(y) permite recorrer la columna en sentido descendente.
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

      for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
      {
        for(int k=0; k<tam_coste; k++)
        {
          // A partir del valor de la columna anexa, se actualiza el
          // valor de la fila inferior de la cada ventana.

          coste_fila[k+(y+rad_vent)*tam_coste]=
               coste_fila[k+(y+rad_vent)*tam_coste]+
               AD[k+(y+rad_vent)*tam_coste+(x+rad_vent)*tam_y*tam_coste]-
               AD[k+(y+rad_vent)*tam_coste+(x-(rad_vent+1))*tam_y*tam_coste];

          coste[k]=coste[k]+coste_fila[k+(y+rad_vent)*tam_coste]-
               coste_fila[k+(y-(rad_vent+1))*tam_coste];

          if(coste[k]<coste[disp_izda[x+(tam_x*y)]])
            { disp_izda[x+(tam_x*y)]=k; }

          if(coste[k]<coste_dcha[k+x+(tam_x*y)])
          {
            disp_dcha[k+x+(tam_x*y)]=k;
            coste_dcha[k+x+(tam_x*y)]=coste[k];
          }
        }
      }

      free(coste);
    }
      free(coste_dcha);
      free(coste_fila);
      free(AD);
}
```

## VERSIÓN 5 · CÁLCULO DEL MAPA DE DISPARIDAD MEDIANTE LA SUPERPOSICIÓN DE VENTANAS

```
////////////////////////////////////////////////////////////////////////

VERSIÓN 6 · CÁLCULO DEL MAPA DE DISPARIDAD MEDIANTE LA SUPERPOSICIÓN
DE VENTANAS.
Este algoritmo es una evolución de la Versión 3. La primera columna y
la fila superior de la imagen (las primeras ventanas de búsqueda de
cada columna) se calculan del mismo modo que en la versión precedente.

La SAD del resto de las ventanas de búsqueda de la imagen se calcula
mediante superposición. . A partir del coste calculado para las tres
ventanas circundantes, se obtiene el valor de la nueva ventana(y,x):
     coste_ventana(y,x)=coste_ventana(y-1,x)+coste_ventana(y,x-1)-
       coste_ventana(y-1,x-1)+AD_píxel(arriba,izquierda)-
       AD_píxel(arriba,derecha)-AD_píxel(abajo,izquierda)+
       AD_píxel(abajo,derecha)
Donde la posición de los píxeles se refiere al conjunto de las cuatro
ventanas de búsqueda superpuestas.

Además, para evitar el cálculo redundante de AD, se calculan todas las
diferencias necesarias al comienzo de la función, almacenándose en el
vector AD.

////////////////////////////////////////////////////////////////////////

void disparidad(unsigned char* im_izda, unsigned char* im_dcha,
                unsigned char* disp_izda, unsigned char* disp_dcha)
{
  // Se definen las matrices de datos:
  //  - coste contiene, a su vez, dos submatrices.
  //     - En la primera se almacenan los resultados de comparar una
  //       columna impar de la imagen izquierda con la correspondiente
  //       de la derecha. Cada columna contiene la comparación del
  //       píxel izquierdo (y,x) con el derecho (y,x) y posteriores,
  //       hasta (y,x+tam_coste). Cada fila de la submatriz almacena
  //       los datos correspondientes a esa misma fila de la imagen.
  //     - En la segunda submatriz se almacenan los mismos datos para
  //       las columnas pares de la imagen.
  //  - coste_dcha almacena los resultados de comparar la imagen
  //     derecha con la izquierda. Cada celda de la matriz guarda la
  //     menor AD encontrada para esa posición.
  //  - AD almacena las diferencias absolutas requeridas a lo largo
  //     del algoritmo.
  //       AD[k+y*tam_coste+x*tam_y*tam_coste]
  //     La matriz AD es un conjunto de submatrices, cada una de las
  //     cuales contiene la información de una columna. En cada una de
  //     estas submatrices se almacena, en sentido x, la AD entre el
  //     píxel izquierdo (y,x) y el derecho (y,x), y sucesivos, hasta
  //     (y,x+tam_coste). En sentido y, recorre todos los elementos de
  //     la columna.

  int* coste=(int*) calloc((tam_coste*tam_y*2), sizeof(int));
  int* coste_dcha=(int*)calloc(tam_x*tam_y, sizeof(int));
  int* AD=(int*)calloc(tam_coste*tam_x*tam_y, sizeof(int));

  // Para posibilitar que la matriz coste_dcha guarde la menor AD
  // encontrada para cada posición, se inicializa en el mayor valor
  // posible.
```

```
   int lim = 255*(2*rad_vent+1)*(2*rad_vent+1);
   for(int d=0; d<(tam_x*tam_y); d++)
   { coste_dcha[d]=lim; }

   // CÁLCULO DE LOS AD.

   // A continuación se calculan todas las AD requeridas a lo largo del
   // algoritmo.
   //  - for(j) y for(i) recorren todos los píxeles de la matriz.
   //  - for(k) permite comparar la ventana izquierda (y,x) con la
   //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

   for (int j=0; j<tam_y; j++)
   {
      for (int i=0; i<tam_x; i++)
      {
         for (int k=0; k<tam_coste; k++)
         {
            AD[k+j*tam_coste+i*tam_y*tam_coste]=
                 AD[k+j*tam_coste+i*tam_y*tam_coste]+
                 abs(im_izda[i+tam_x*j]-im_dcha[k+i+tam_x*j]);
         }
      }
   }

// Primera columna · Primer elemento.

      // A continuación se calculan las SAD del primer elemento de la
      // primera columna de la imagen izquierda.
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
      //  - for(j) y for(i) recorren todos los píxeles de la ventana.
      // El resultado de acumular todas las AD de una fila de la ventana
      // de búsqueda se almacena en coste_fila.
      // El resultado de la ventana de búsqueda completa se almacena en
      // coste.

      for(int k=0; k<tam_coste; k++)
      {
         for(int j=-rad_vent; j<=rad_vent; j++)
         {
            for(int i=-rad_vent; i<=rad_vent; i++)
            {
               coste[k+rad_vent*tam_coste]=coste[k+rad_vent*tam_coste]+
                    AD[k+(rad_vent+j)*tam_coste+(rad_vent+i)*tam_y*tam_coste];
            }
         }

         // En disp_izda se almacena la menor disparidad izquierda
         // encontrada hasta el momento para cada píxel. Este valor se
         // compara con cada SAD a medida que se calculan, guardando el
         // nuevo valor en caso de ser menor.

         if(coste[k+rad_vent*tam_coste]<
            coste[disp_izda[rad_vent+(tam_x*rad_vent)]+rad_vent*tam_coste])

           { disp_izda[rad_vent+(tam_x*rad_vent)]=k; }

         // En coste_dcha se almacena el menor valor de SAD encontrado
         // hasta el momento para cada píxel. Este valor se compara con
```

```
        // cada SAD a medida que se calculan, guardando el nuevo valor
        // en caso de ser menor. En disp_dcha se almacena, de forma
        // simultánea, la disparidad a la que corresponde el valor
        // guardado en coste_dcha.

        if(coste[k+rad_vent*tam_coste]<
           coste_dcha[k+rad_vent+(tam_x*rad_vent)])

        {
          disp_dcha[k+rad_vent+(tam_x*rad_vent)]=k;
          coste_dcha[k+rad_vent+(tam_x*rad_vent)]=
               coste[k+rad_vent*tam_coste];
        }
      }

// Primera columna · Demás elementos.

      // A continuación se calculan las SAD de los demás elementos de la
      // primera columna de la imagen izquierda.
      //  - for(y) permite recorrer la columna en sentido descendente.
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).
      //  - for(i) recorre el borde inferior de la ventana de búsqueda,
      //     calculando la nueva fila implicada.

      for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
      {
        for(int k=0; k<tam_coste; k++)
        {
          // El valor de coste para la ventana y de la primera columna
          // se calcula a partir del coste de la ventana previa (y-1).

          coste[k+y*tam_coste]=coste[k+(y-1)*tam_coste];

          for(int i=-rad_vent; i<=rad_vent; i++)
          {
            // Al valor de coste almacenado para la ventana precedente,
            // se le resta el valor de la fila superior (que sólo
            // pertenece a la anterior ventana) y se le suma el valor
            // de la fila inferior (privativa de la nueva).

            coste[k+y*tam_coste]=coste[k+y*tam_coste]+
                 AD[k+(y+rad_vent)*tam_coste+(rad_vent+i)*tam_y*tam_coste]-
                 AD[k+(y-(rad_vent+1))*tam_coste+
                    (rad_vent+i)*tam_y*tam_coste];
          }

          if(coste[k+y*tam_coste]<
             coste[disp_izda[rad_vent+(tam_x*y)]+y*tam_coste])

            { disp_izda[rad_vent+(tam_x*y)]=k; }

          if(coste[k+y*tam_coste]<coste_dcha[k+rad_vent+(tam_x*y)])
          {
            disp_dcha[k+rad_vent+(tam_x*y)]=k;
            coste_dcha[k+rad_vent+(tam_x*y)]=coste[k+y*tam_coste];
          }
        }
      }
```

```
   // Para poder discriminar si la actual ventana de búsqueda pertenece
   // a una columna par o impar, y por tanto, hacer uso de una parte u
   // otra de la memoria, se definen dos variables booleanas. Para las
   // columnas pares las variables tomarán los valores de inicio (notar
   // que la primera columna, impar, se ha calculado previamente).

   bool par=1;
   bool impar=0;

   // for(x) recorre todas las columnas de la imagen de izquierda a
   // derecha.

   for (int x=(rad_vent+1); x<(tam_x-tam_coste); x++)
   {

// Demás columnas · Primer elemento.

     for(int k=0; k<tam_coste; k++)
     {
       // El valor de coste para una ventana x de la primera fila se
       // calcula a partir del coste de la ventana de la columna anexa
       // (x-1).

       coste[k+rad_vent*tam_coste+par*tam_y*tam_coste]=
            coste[k+rad_vent*tam_coste+impar*tam_y*tam_coste];

       // for(j) recorre los bordes laterales de la ventana de
       // búsqueda. Al valor almacenado (correspondiente a la columna
       // anexa) se le resta la AD del píxel izquierdo, sumándole la
       // del derecho para obtener el valor de la fila de la nueva
       // columna.

       for(int j=-rad_vent; j<=rad_vent; j++)
       {
         coste[k+rad_vent*tam_coste+par*tam_y*tam_coste]=
              coste[k+rad_vent*tam_coste+par*tam_y*tam_coste]+
              AD[k+(rad_vent+j)*tam_coste+(x+rad_vent)*tam_y*tam_coste]-
              AD[k+(rad_vent+j)*tam_coste+
                (x-(rad_vent+1))*tam_y*tam_coste];
       }

       if(coste[k+rad_vent*tam_coste+par*tam_y*tam_coste]<
            coste[disp_izda[x+(tam_x*rad_vent)]+rad_vent*tam_coste+
                  par*tam_y*tam_coste])

         { disp_izda[x+(tam_x*rad_vent)]=k; }

       if(coste[k+rad_vent*tam_coste+par*tam_y*tam_coste]<
          coste_dcha[k+x+(tam_x*rad_vent)])

       {
         disp_dcha[k+x+(tam_x*rad_vent)]=k;
         coste_dcha[k+x+(tam_x*rad_vent)]=
              coste[k+rad_vent*tam_coste+par*tam_y*tam_coste];
       }
     }

// Demás columnas · Demás elementos.

     // A continuación se calculan las SAD de los demás elementos.
     //  - for(y) permite recorrer la columna en sentido descendente.
```

```
      //  - for(k) permite comparar la ventana izquierda (y,x) con la
      //     ventana derecha(y,x) y posteriores, hasta (y,x+tam_coste).

      for (int y=(rad_vent+1); y<(tam_y-rad_vent); y++)
      {
        for(int k=0; k<tam_coste; k++)
        {
          // El coste de la ventana (y,x),por superposición, sería:
          // coste(y,x)=coste(y-1,x)+coste(y,x-1)-coste(y-1,x-1)+
          //   px(arriba,izquierda)-px(arriba,derecha)-
          //   px(abajo,izquierda)+px(abajo,derecha)
          // Donde la posición de los píxeles se refiere al conjunto de
          // las cuatro ventanas de búsqueda superpuestas.
          // (Notar que el orden de la fórmula coincide con el descrito
          // en el algoritmo).

          coste[k+y*tam_coste+par*tam_y*tam_coste]=
               coste[k+(y-1)*tam_coste+par*tam_y*tam_coste]+
               coste[k+y*tam_coste+impar*tam_y*tam_coste]-
               coste[k+(y-1)*tam_coste+impar*tam_y*tam_coste]+
               AD[k+(y-(rad_vent+1))*tam_coste+
                  (x-(rad_vent+1))*tam_y*tam_coste]-
               AD[k+(y-(rad_vent+1))*tam_coste+
                  (x+(rad_vent))*tam_y*tam_coste]-
               AD[k+(y+(rad_vent))*tam_coste+
                  (x-(rad_vent+1))*tam_y*tam_coste]+
               AD[k+(y+(rad_vent))*tam_coste+
                  (x+(rad_vent))*tam_y*tam_coste];

          if(coste[k+y*tam_coste+par*tam_y*tam_coste]<
              coste[disp_izda[x+(tam_x*y)]+y*tam_coste+par*tam_y*tam_coste])

            { disp_izda[x+(tam_x*y)]=k; }

          if(coste[k+y*tam_coste+par*tam_y*tam_coste]<
              coste_dcha[k+x+(tam_x*y)])

          {
            disp_dcha[k+x+(tam_x*y)]=k;
            coste_dcha[k+x+(tam_x*y)]=
                 coste[k+y*tam_coste+par*tam_y*tam_coste];
          }
        }
      }

      par=par-1;
      impar=impar-1;
    }
    free(coste);
    free(coste_dcha);
    free(AD);
}
```